Numerical Algorithms

Simple Algorithms to speed up basic functions, using these techniques can optimize the basic functions so that you can focus on the main algorithm.

Things to be covered

- Euclid's Algorithm
- Least common multiple
- Prime testing by trial division
- Sieve of Eratosthenes
- Horner's rule
- Factoring
- Efficient exponentation

Euclid's Algorithm (GCD)

- The algorithm is used to obtain the GCD of any two given numbers
- By continuoesly calculating the remainder of the two numbers, the GCD is determined as soon as the remainder eqauls 0

Euclid's Pseudo code

```
GCD(int a,int b)
    if b == 0
        return a
    else
        return GCD(b,a%b)
```

Least common multiple

- As soon as you understand GCD it can be applied to finding the least common multiple
- The method is derived from the High School method of calculating the prime factors of both numbers then multiplying the union of each number

Least common multiple

Take 24 and 36

24 = 2.2.2.3
36 = 2.2. .3.3

Union = 2.2.2.3.3

LCM = 72

Note that the it can be simplified to:
LCM = (24.36)/GCD(36,24)

thus LCM = (a*b)/GCD(a,b)

Prime testing by trail division

- Note that you would only use this method to test whether a given number is prime
- To generate primes use Sieve of Eratosthenes
- Note: You only need to test upto $\sqrt{N}$
- This can be optimised by testing 2 apart then use an interval of 2
- $O(\sqrt{N})$

Sieve of Eratosthenes

- Generates a list of primes
- Calculates primes in a range from 2 to N
- Faster than repeated trail division
- Start by assuming all numbers except 1 are prime

The Algorithm

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Iterate through the numbers in increasing order until you find a number that is marked as prime

The Algorithm

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Confirm the number as prime then mark the multiples of 2 onwards from 2^2 as not prime

The Algorithm

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Now continue using the same pattern

The Algorithm

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

As soon as you finish with 7 there is no more need to eliminate as 11^2 > 100

The Algorithm

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Green primes

Pseudo Code

```
Sieve(int n)
    bool pTest[n+1]
    //Set values == True
    for i = 2 to n
        if pTest[i]
            //Add to list
            for j = i*i to n step i
                pTest[j] = False
    return list
```

Horner's rule

- An efficient way to calculate polynomials
- Take $f(X)=5X^4+12X^3-2X^2-2X+4$
- This can become $f(X)=X(X(X(5X+12)-2)-2)+4$
- By using the notation above this can be reduced to 8 operations compared to 14 in the first
- Thus you can use Horner's rule for a polynomials to the Nth degree in the form of:

$$f(X)=A_0X^N+A_1X^{N-1}-A_3X^{N-2}...+A_{N-1}X+A_N$$

Pseudo Code

```
Horner(double [ ] A,double X,int N)
    float Ans = A[0]
    for i = 1 to N
        Ans *= X
        Ans += A[i]
    return Ans
```

Integer Factoring

- When you need to reduce numbers to their prime factors
- DON'T generate a list of primes
- Starting with 2 and moving upwards will ensure all numbers are prime

Pseudo Code

```
PrimeFactors(int N)
    Ans = N
    array Factors
    for i = 2 to N
        while (Ans % i == 0)
            Factors.append(i)
            Ans /= i
        if (Ans == 1) break
    return Factors
```

Efficient Exponentation

- Calculate a^b in O(log b) time
- There are two methods, both are based on the binary representation of the exponent
- Left to Right (Recursive overhead)
- Right to Left (No recursive overhead)
- Both methods are O(log b)

Left to Right

- Take the statement a^{29}
- That can be represented as a^{11101_2}
- Initialize an answer variable to 1
- Then start from the left most value
- If the value is 1 multiply the answer variable with a
- Move to the next position and square the answer

Left to Right Pseudo Code

```
LeftToRight(int a,int b)
    if (b == 0)  //exit statement
        return 1;
    else
        if (b % 2 == 1)
            return a*LeftToRight(a,b/2)**2
        else
            return LeftToRight(a,b/2)**2
```

Right to Left

- Similar to Left to Right, but doesn't need recursion
- You keep an additional index of the value of the exponent at the current position of the binary representation
- If the value is 1 at that position, multiply the answer with the index

Pseudo Code

```
RightToLeft(int a, int b)
    int Index = a
    int Answer = 1
    while (b)
        if (b % 2 == 1)
            Answer *= Index
        Index *= Index
        b /= 2
    return Answer
```

Questions

?