Chapter 2

Algorithm Complexity

Design challenges

- Designing an algorithmic solution is (almost) good:
  - The algorithm should not take “ages”
  - It should not consume “too much” memory.
- In this chapter, we will discuss the following:
  - How to estimate the time required for a program.
  - How to reduce the running time of a program from days or years to fractions of a second.
  - The results of careless use of recursion.
  - Very efficient algorithms to raise a number to a power and to compute the greatest common divisor of two numbers.

Efficient Algorithms

- A city has n view points
- Buses move from one view point to another
- A bus driver wishes to follow the shortest path (travel time wise).
- Every view point is connected to another by a road.
- However, some roads are less congested than others.
- Also, roads are one-way, i.e., the road from view point 1 to 2, is different from that from view point 2 to 1.

- How to find the shortest path between any two pairs?
- Naïve approach:
  - List all the paths between a given pair of view points
  - Compute the travel time for each.
  - Choose the shortest one.
- How many paths are there between any two view points (without revisits)?

$$n! \cong (n/e)^n$$

- It will be impossible to run the algorithm for n = 30

What is efficiency of an algorithm?

- Run time in the computer is machine-dependent
- Example: Multiply two positive integers a and b
- Subroutine 1: Multiply a and b
- Subroutine 2:

```
v = a,     w =  b
While   w > 1
        { v  = v + a;
          w = w - 1}
 Output v
```

Machine-Dependent Analysis

- First subroutine has 1 multiplication.
- Second subroutine has b additions and subtractions.
- For some architectures, 1 multiplication is more expensive than b additions and subtractions.
- Ideally:
  - programme all alternative algorithms
  - run them on the target machine
  - find which is more/most efficient!

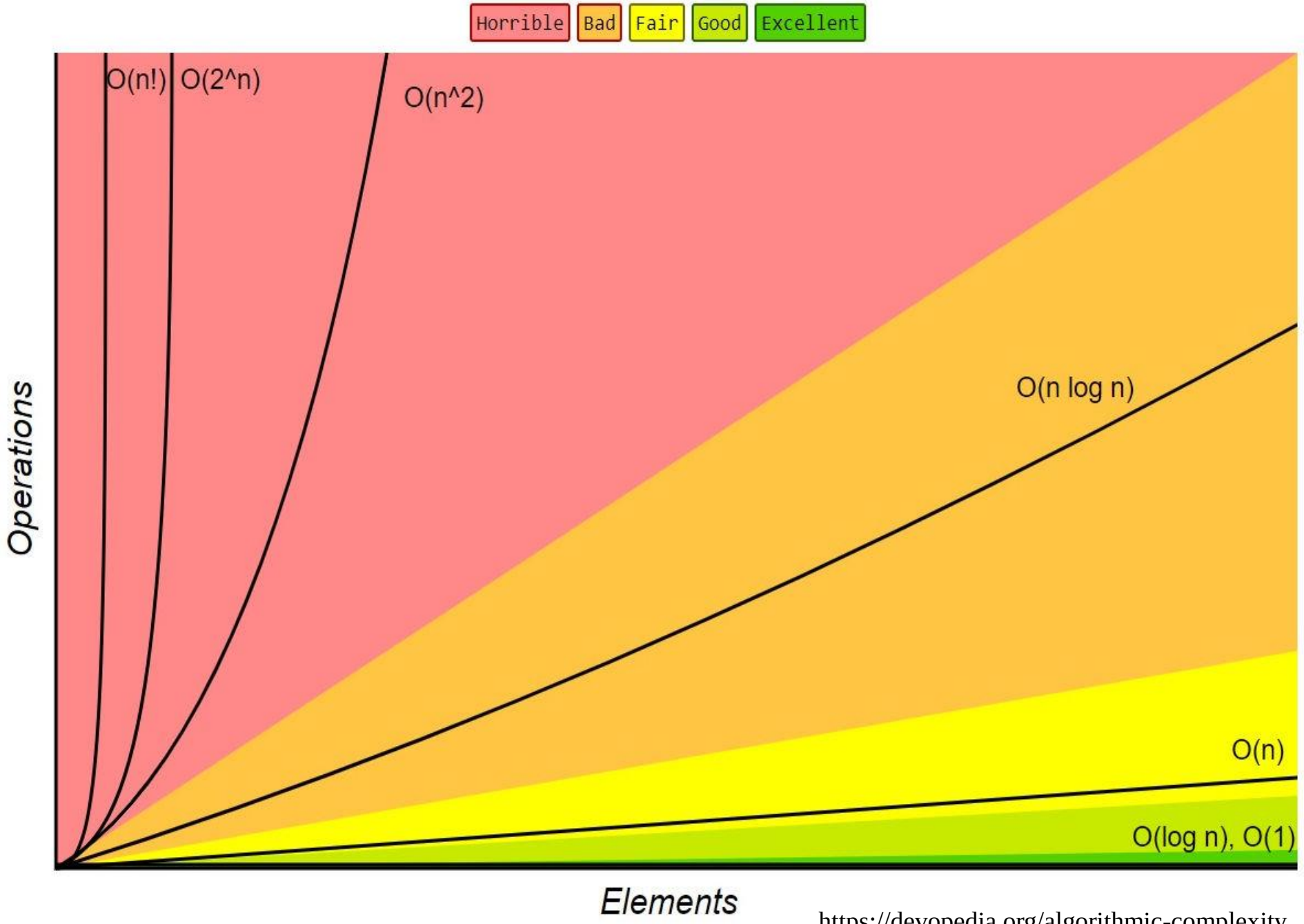
Machine-Independent Analysis

- We will assume that every basic operation takes constant time
- Example of Basic Operations:
  - Addition, Subtraction, Multiplication, Memory Access
- Non-basic Operations:
  - Sorting, Searching
- Efficiency of an algorithm is thus measured in terms of the number of basic operations it performs
  - We do not distinguish between the basic operations.

- Subroutine 1 uses 1 basic operation
- Subroutine 2 uses 2b basic operations

  ➔ Subroutine 1 is more efficient.

- This measure is good for all large input sizes
- **In fact**, we will not worry about the exact values number of operations, but will look at ``broad classes' of values.
  - Let there be n inputs.
  - If an algorithm needs n basic operations and another needs 2n basic operations, we will consider them to be in the same efficiency category.
  - However, we distinguish between exp(n), n, log(n)
- We worry about the speed of our algorithms for large input sizes.



https://devopedia.org/algorithmic-complexity

Weak ordering

Consider the following definitions:

- We will consider two functions to be equivalent, $f \sim g$, if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$

  where $0 < c < \infty$

- We will state that $f < g$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

For functions we are interested in, these define a weak ordering

Weak ordering

Let $f(n)$ and $g(n)$ describe the run-time of two algorithms. In general, there are functions s.t.:

- If $f(n) \sim g(n)$, then it is always possible to improve the performance of one function over the other by purchasing a faster computer
- If $f(n) < g(n)$, then you can never purchase a computer fast enough so that the second function always runs in less time than the first

- Note that for small values of n, it may be reasonable to use an algorithm that is asymptotically more expensive, but we will consider these on a one-to-one basis

Function Orders

Definition 2.1

$f(n) = O(g(n))$ if there are positive *constants* c and n_0 such that $0 \leq f(n) \leq c * g(n)$ when $n \geq n_0$

i.e. A function *f(n)* is *O(g(n))* if rate of growth of *f(n)* is not faster than that of *g(n)*.

i.e. if $lim_{n \to \infty} f(n)/g(n)$ exists and is finite, then *f(n)* is *O(g(n))*

Intuitively, (not exactly) *f(n)* is *O(g(n))* means $f(n) \leq g(n)$ for all *n* beyond some value n_0; i.e. *g(n)* is an *upper bound* for *f(n)*.

Remark

Before we begin, let us make some assumptions:

- Our functions will describe the time or memory required to solve a problem of size n
- Restrictions are put on the functions:
  - They are defined for $n \geq 0$
  - They are strictly positive for all n
    - In fact, $f(n) > c$ for some value $c > 0$
    - That is, any problem requires at least one instruction and byte
  - They are increasing (monotonic increase)

Examples of Functions

sqrt(n) , n, 2n, ln n, exp(n), n + sqrt(n) , n + n^2

$\lim_{n\to\infty}$ sqrt(n) /n = 0,	sqrt(n) is O(n)
$\lim_{n\to\infty}$ n/sqrt(n) = infinity,	n is not O(sqrt(n))
$\lim_{n\to\infty}$ n /2n = 1/2,	n is O(2n)
$\lim_{n\to\infty}$ 2n /n = 2,	2n is O(n)

Other examples:

$\lim_{n\to\infty} \ln(n)/n = 0$ — ln(n) is O(n)

$\lim_{n\to\infty} n/\ln(n) = \text{infinity}$ — n is not O(ln(n))

$\lim_{n\to\infty} \exp(n)/n = \text{infinity}$ — exp(n) is not O(n)

$\lim_{n\to\infty} n/\exp(n) = 0$ — n is O(exp(n))

$\lim_{n\to\infty} (n+\text{sqrt}(n))/n = 1$ — n + sqrt(n) is O(n)

$\lim_{n\to\infty} n/(\text{sqrt}(n)+n) = 1$ — n is O(n+sqrt(n))

$\lim_{n\to\infty} (n + n^2)/n = \text{infinity}$ — $n + n^2$ is not O(n)

$\lim_{n\to\infty} n/(n + n^2) = 0$ — n is $O(n + n^2)$

Implication of Big Oh notation

- Suppose we know that our algorithm uses at most $O(f(n))$ basic steps for any n inputs, and n is sufficiently large
- then we know that our algorithm will terminate after executing at most $f(n)$ basic steps.
- We also know that a basic step takes a constant time in a machine.

➔ Our algorithm will terminate in $f(n)$ units of time, for all large n.

Other Complexity Notation

Now a lower bound notation, $\Omega(n)$

Definition 2.2

$f(n) = \Omega(g(n))$ if there are positive *constants c* and n_0 such that $f(n) \geq c * g(n)$ when $n \geq n_0$.

$lim_{n \to \infty} (f(n)/g(n)) > 0$, if $lim_{n \to \infty} (f(n)/g(n))$ exists

$g(n)$ is a *lower bound* on $f(n)$

Implication of the Ω Notation

Suppose, an algorithm has complexity Ω(f(n)) .

This means that there exists a positive constant c such that for all sufficiently large n, there exists at least one input for which the algorithm consumes at least c * f(n) steps.

Other Complexity Notation

Definition 2.3

$f(n) = \theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

- $\theta(g(n))$ is referred to as ``asymptotic equality''
- $lim_{n \to \infty} (f(n)/g(n))$ is a finite, positive constant, if it exists

f(n) has a rate of growth equal to that of g(n)

Other Complexity Notation

Definition 2.4

$f(n) = o(g(n))$ if, for all positive constants c, there exists an n_0 such that $f(n) < c * g(n)$ when $n > n_0$. (``asymptotic strict inequality'')

- Less formally, $f(n) = o(p(n))$ if $f(n) = O(g(n))$ and $f(n) \neq \theta(g(n))$.
- *f(n)* is *o(g(n))* if given any positive constant *c*, there exists some m such that $f(n) < c * g(n)$ for all $n \geq m$

  $lim_{n \to \infty} (f(n)/g(n)) = 0$, if $lim_{n \to \infty} (f(n)/g(n))$ exists

- Small *oh* means: ""is ultimately smaller than"

Rates of gowth

Suppose that $f(n)$ and $g(n)$ satisfy $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ where $0 < c < \infty$, it follows that $f(n) = \Theta(g(n))$

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$ where $0 \le c < \infty$, it follows that $f(n) = O(g(n))$

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, we will say $f(n) = o(g(n))$

$f(n)$ has a rate of growth less than that of $g(n)$

Terminology

Asymptotically less than or equal to	**O**
Asymptotically greater than or equal to	**Ω**
Asymptotically equal to	**θ**
Asymptotically strictly less	**o**

Recap

$$f(n) = \mathbf{o}(g(n)) \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$$f(n) = \mathbf{O}(g(n)) \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

$$f(n) = \mathbf{\Theta}(g(n)) \qquad 0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$$

$$f(n) = \mathbf{\Omega}(g(n)) \qquad \lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$$

Example Functions

sqrt(n) , n, 2n, ln n, exp(n), n + sqrt(n) , $n + n^2$

$\lim_{n\to\infty}$ sqrt(n) /n = 0 sqrt(n) is o(n), O(n)

$\lim_{n\to\infty}$ n/sqrt(n) = infinity n is Ω(sqrt(n))

$\lim_{n\to\infty}$ n/ln(n) = infinity n is Ω(ln(n))

$\lim_{n\to\infty}$ 2n /n = 2 2n is θ(n)

$\lim_{n\to\infty}$ n /2n = 1/2 n is θ(2n)

Terminology

The most common classes are given names:

$\Theta(1)$	constant
$\Theta(\ln(n))$	logarithmic
$\Theta(n)$	linear
$\Theta(n \ln(n))$	"$n \log n$"
$\Theta(n^2)$	quadratic
$\Theta(n^3)$	cubic
2^n, e^n, 4^n, ...	exponential

Little-o as a Weak Ordering

We can show that, for example

$$\ln(n) = \mathbf{o}(n^p)$$

for any $p > 0$

Proof: Using l'Hôpital's rule, we have

$$\lim_{n\to\infty} \frac{\ln(n)}{n^p} = \lim_{n\to\infty} \frac{1/n}{pn^{p-1}} = \lim_{n\to\infty} \frac{1}{pn^p} = \frac{1}{p} \lim_{n\to\infty} n^{-p} = 0$$

Little-o as a Weak Ordering

Other observations:

- If p and q are real positive numbers where $p < q$, it follows that

$$n^p = \mathbf{o}(n^q)$$

- For example, matrix-matrix multiplication is $\Theta(n^3)$ but a refined algorithm is $\Theta(n^{\lg(7)})$ where $\lg(7) \approx 2.81$

- Also, $n^p = \mathbf{o}(\ln(n)n^p)$, but $\ln(n)n^p = \mathbf{o}(n^q)$
  - n^p has a slower rate of growth than $\ln(n)n^p$, but
  - $\ln(n)n^p$ has a slower rate of growth than n^q for $p < q$

Algorithms Analysis

An algorithm is said to have *polynomial time complexity* if its run-time may be described by $O(n^d)$ for some fixed $d \geq 0$

- We will consider such algorithms to be efficient

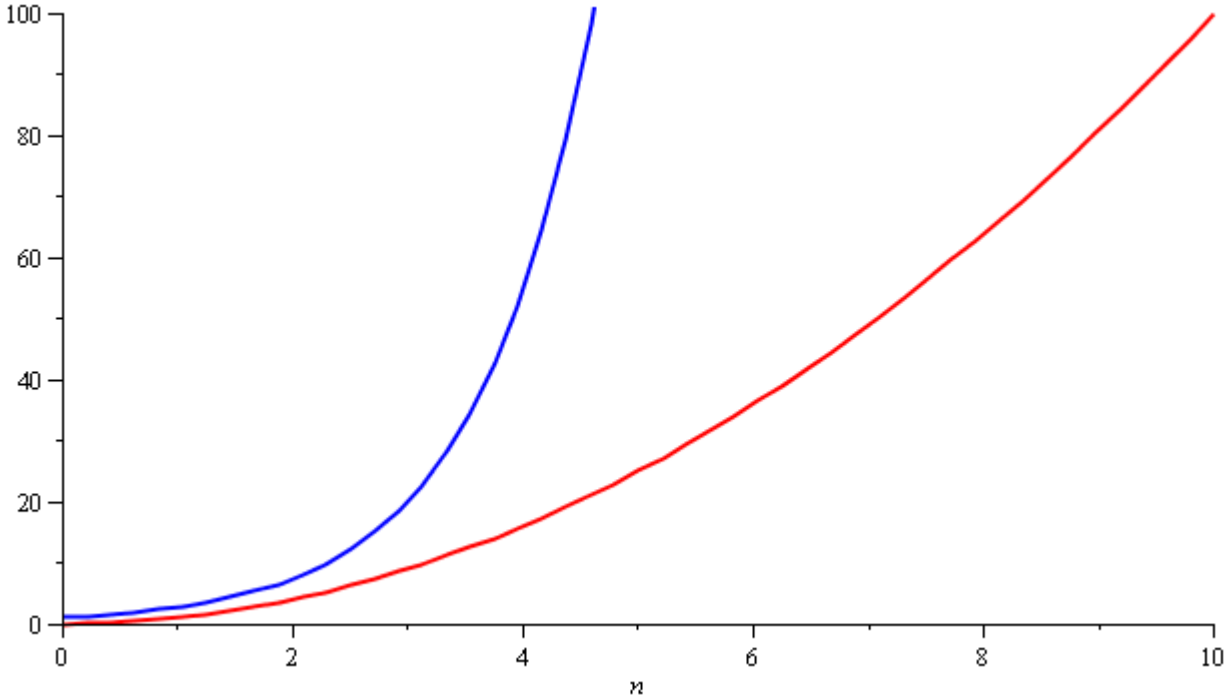
Problems that have no known polynomial-time algorithms are said to be intractable

- Traveling salesman problem: find the shortest path that visits n cities
- Best run time: $\Theta(n^2 2^n)$

Algorithm Analysis

In general, you don't want to implement exponential-time or exponential-memory algorithms

– Warning: don't call a quadratic curve "exponential", either...please



Rules for arithmetic with big-O symbols

- If $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$, then
  - (a) $T_1(n) + T_2(n) = O(f(n) + g(n))$ (intuitively and less formally it is $O(\max(f(n), g(n)))$),
  - (b) $T_1(n) * T_2(n) = O(f(n) * g(n))$.
- If $T(n)$ is a polynomial of degree k, then $T(n) = \theta(n^k)$.
- $log^k\ n = O(n)$ for any constant k.

  This tells us that logarithms grow very slowly.

Rules for arithmetic with big-O symbols

- If $f(n) = O(g(n))$, then $c * f(n) = O(g(n))$ for any constant c.
- If $f_1(n) = O(g(n))$ but $f_2(n) = o(g(n))$, then $f_1(n) + f_2(n) = O(g(n))$.
- If $f(n) = O(g(n))$, and $g(n) = o(h(n))$, then $f(n) = o(h(n))$. (complexity of $f \circ g$)
- These are not all of the rules, but they're enough for most purposes.

Complexity of a Problem vs Algorithm

A problem is O(f(n)) means there is some O(f(n)) algorithm to solve the problem.

A problem is Ω(f(n)) means every algorithm that can solve the problem is Ω(f(n))

Algorithm Complexity Analysis

- We define $T\text{avg}(N)$ and $T\text{worst}(N)$, as the average and worst-case running time, resp., used by an algorithm on input of size N. Clearly, $T\text{avg}(N) \leq T\text{worst}(N)$.
- Occasionally, the best-case performance of an algorithm is analyzed.
  - of little interest: does not represent the typical behavior.
- Average-case performance often reflects typical behavior
- **Worst-case performance represents a guarantee for performance on any possible input**
- Interested in algorithm analysis not programme analysis: implementation issues/details/inefficiencies, etc.

Algorithm Complexity Analysis

Consider the following algorithm

```
diff = sum = 0;
For (k=0: k < N; k++)
        sum → sum + 1;
        diff → diff - 1;
For (k=0: k < 3N; k++)
        sum → sum - 1;
```

- First line takes 2 basic steps
- Every iteration of first loop takes 2 basic steps.
- First loop runs N times
- Every iteration of second loop takes 1 basic step
- Second loop runs for 3N times
- Overall, 2 + 2N + 3N steps
- This is O(N)

Rules

Complexity of a loop:

O(Number of iterations in a loop * maximum complexity of each iteration)

Nested Loops:

Analyze the innermost loop first,

Complexity of next outer loop = number of iterations in this loop * complexity of inner loop, etc.

```
sum = 0;
For (i=0; i < N; i++)
    For (j=0; j < N; j++)   sum → sum + 1;
```

Inner loop: O(N)

Outer loop: N iterations

Overall: $O(N^2)$

```
for( i = 0; i < n; ++i )
    a[ i ] = 0;
for( i = 0; i < n; ++i )
    for( j = 0; j < n; ++j )
        a[ i ] += a[ j ] + i + j;
```

First loop: O(N)

Inner loop: O(N)

Outer loop: N iterations

Overall: $O(N^2) + O(N)$
SO O(N)

If (Condition)

S1

Else S2

Maximum of the two complexities

If (yes)

print(1,2,….1000N)

else print(1,2,….N^2)

Analysis of recursion

- Suppose we have the code (not a good one):

```
Long fib (int n) {
        if (n <= 1)                                  1
                return 1;                            2
        else
                return fib(n – 1) + fib(n – 2);      3
}
```

T(0) = T(1) = 1;

n>=2 T(n) = cost of constant op at 1 + cost of line 3 work

T(n) = 1 op + (addition + 2 function calls) = O(1) + (addition + cos of fib(n-1) + cost fib(n-2)) = 2 + T(n-1) + T(n-2)

Analysis of recursion

- Easy to show by induction T(n) >= fib(n)
- Can prove for n>4, fib(n) >= $(3/2)^n$

➔Running time of the programme grows exponentially

By using an array and a for loop, the programme running time can be reduced substantially.

Maximum Subsequence Problem

Given an array of N elements

Need to find i, j such that the sum of all elements between the ith and jth positions is maximum for all such sums

Running time of 4 algorithms for max subsequence sum (in seconds) [Weiss, Fig 2.2]

Input Size	Algorithm Time 1 $O(N^3)$	2 $O(N^2)$	3 $O(N \log N)$	4 $O(N)$
$N = 100$	0.000159	0.000006	0.000005	0.000002
$N = 1{,}000$	0.095857	0.000371	0.000060	0.000022
$N = 10{,}000$	86.67	0.033322	0.000619	0.000222
$N = 100{,}000$	NA	3.33	0.006700	0.002205
$N = 1{,}000{,}000$	NA	NA	0.074870	0.022711

Algorithm 1

```
/**
 * Cubic maximum contiguous subsequence sum algorithm.
 */
int maxSubSum1( const vector<int> & a )
{
    int maxSum = 0;

    for( int i = 0; i < a.size( ); ++i )
        for( int j = i; j < a.size( ); ++j )
        {
            int thisSum = 0;

            for( int k = i; k <= j; ++k )
                thisSum += a[ k ];

            if( thisSum > maxSum )
                maxSum = thisSum;
        }

    return maxSum;
}
```

Analysis of Algorithm 1

Inner loop:

$$\sum_{j=i}^{N-1} (j-i+1) = (N-i+1)(N-i)/2$$

Outer Loop:

$$\sum_{i=0}^{N-1} (N-i+1)(N-i)/2 = (N^3 + 3N^2 + 2N)/6$$

Overall: $O(N^3)$

Note: The innermost loop can be made more efficient leading to $O(N^2)$ ➔ exercise with complexity analysis

Algorithm 2

```
/**
 * Quadratic maximum contiguous subsequence sum algorithm.
 */
int maxSubSum2( const vector<int> & a )
{
    int maxSum = 0;

    for( int i = 0; i < a.size( ); ++i )
    {
        int thisSum = 0;
        for( int j = i; j < a.size( ); ++j )
        {
            thisSum += a[ j ];

            if( thisSum > maxSum )
                maxSum = thisSum;
        }
    }

    return maxSum;
}
```

Divide and Conquer

- Break a big problem into two small sub-problems
- Solve each of them efficiently.
- Combine the two solutions

Maximum subsequence sum by divide and conquer

- Divide the array into two parts: left part, right part each to be solved recursively
- Max. subsequence lies completely in left, or completely in right or spans the middle.
- If it spans the middle, then it includes the max subsequence in the left ending at the last element and the max subsequence in the right starting from the center

4 –3 5 –2 -1 2 6 -2

Max subsequence sum for first half = 6

second half = 8

Max subsequence sum for first half ending at the last element (4th element included) is 4

Max subsequence sum for second half starting at the first element (5th element included) is 7

Max subsequence sum spanning the middle is 4 + 7 = 11

Max subsequence spans the middle

Algorithm 3: Divide & Conquer

```
 6   int maxSumRec( const vector<int> & a, int left, int right )
 7   {
 8       if( left == right )  // Base case
 9           if( a[ left ] > 0 )
10               return a[ left ];
11           else
12               return 0;
13
14       int center = ( left + right ) / 2;
15       int maxLeftSum  = maxSumRec( a, left, center );
16       int maxRightSum = maxSumRec( a, center + 1, right );
17
18       int maxLeftBorderSum = 0, leftBorderSum = 0;
19       for( int i = center; i >= left; --i )
20       {
21           leftBorderSum += a[ i ];
22           if( leftBorderSum > maxLeftBorderSum )
23               maxLeftBorderSum = leftBorderSum;
24       }
```

```
26          int maxRightBorderSum = 0, rightBorderSum = 0;
27          for( int j = center + 1; j <= right; ++j )
28          {
29              rightBorderSum += a[ j ];
30              if( rightBorderSum > maxRightBorderSum )
31                  maxRightBorderSum = rightBorderSum;
32          }
33
34          return max3( maxLeftSum, maxRightSum,
35                       maxLeftBorderSum + maxRightBorderSum );
36      }
37
38      /**
39       * Driver for divide-and-conquer maximum contiguous
40       * subsequence sum algorithm.
41       */
42      int maxSubSum3( const vector<int> & a )
43      {
44          return maxSumRec( a, 0, a.size( ) - 1 );
45      }
```

Algorithm 3 Analysis

- If N=1; lines 8 to 12 executed; taken to be one unit
  ➔ T(1) = 1
- N>1: 2 recursive calls, 2 for loops, some bookkeeping ops (e.g. lines 14, 34)
  - The 2 for loops (lines 19 to 32): clearly O(N)
  - Lines 8, 14, 18, 26, 34: constant time; ignored compared to O(N)
  - Recursive calls made on half the array size each.

    ➔ 2 * T(N/2)

SO: programme time is 2 * T(N/2) + O(N) with T(1) =1

Complexity Analysis

$T(1)=1$

$T(n) = 2T(n/2) + cn$

$= 2.cn/2 + 4T(n/4) + cn$

$= 4T(n/4) + 2cn$

$= 8T(n/8) + 3cn$

$=\ldots\ldots\ldots\ldots..$

$= 2^iT(n/2^i) + icn$

$=\ldots\ldots\ldots\ldots\ldots\ldots$ (reach a point when $n = 2^i$ $i=\log n$

$= n.T(1) + c\ n \log n$

$n + c\ n \log n = O(n \log n)$

Algorithm 4

```
1   /**
2    * Linear-time maximum contiguous subsequence sum algorithm.
3    */
4   int maxSubSum4( const vector<int> & a )
5   {
6       int maxSum = 0, thisSum = 0;
7
8       for( int j = 0; j < a.size( ); ++j )
9       {
10          thisSum += a[ j ];
11
12          if( thisSum > maxSum )
13              maxSum = thisSum;
14          else if( thisSum < 0 )
15              thisSum = 0;
16      }
17
18      return maxSum;
19  }
```

Algorithm 4 Analysis

- T(N) = O(N) Obvious!
- What is notso obvious is the logic of the algorithm.

Exercise: convince yourself that it does the required work.

Binary Search

- You have a sorted list of numbers
- You need to search the list for a specific number
- If the number exists find its position.
- If the number does not exist you need to detect that

```
Search(num, A[],left, right)
  {
        if (left = right)
          {
              if (A[left ]=num)     return(left) and exit;
              else conclude NOT PRESENT and exit;
           }
     center = ⌊ (left + right)/2 ⌋;
      If (A[center] < num)
          Search(num, A[], center + 1, right);
     If (A[center]>num)
        Search(num, A[], left, center );
    If (A[center]=num) return(center) and exit;
}
```

Complexity Analysis

$T(n) = T(n/2) + c$

$O(\log n)$ complexity