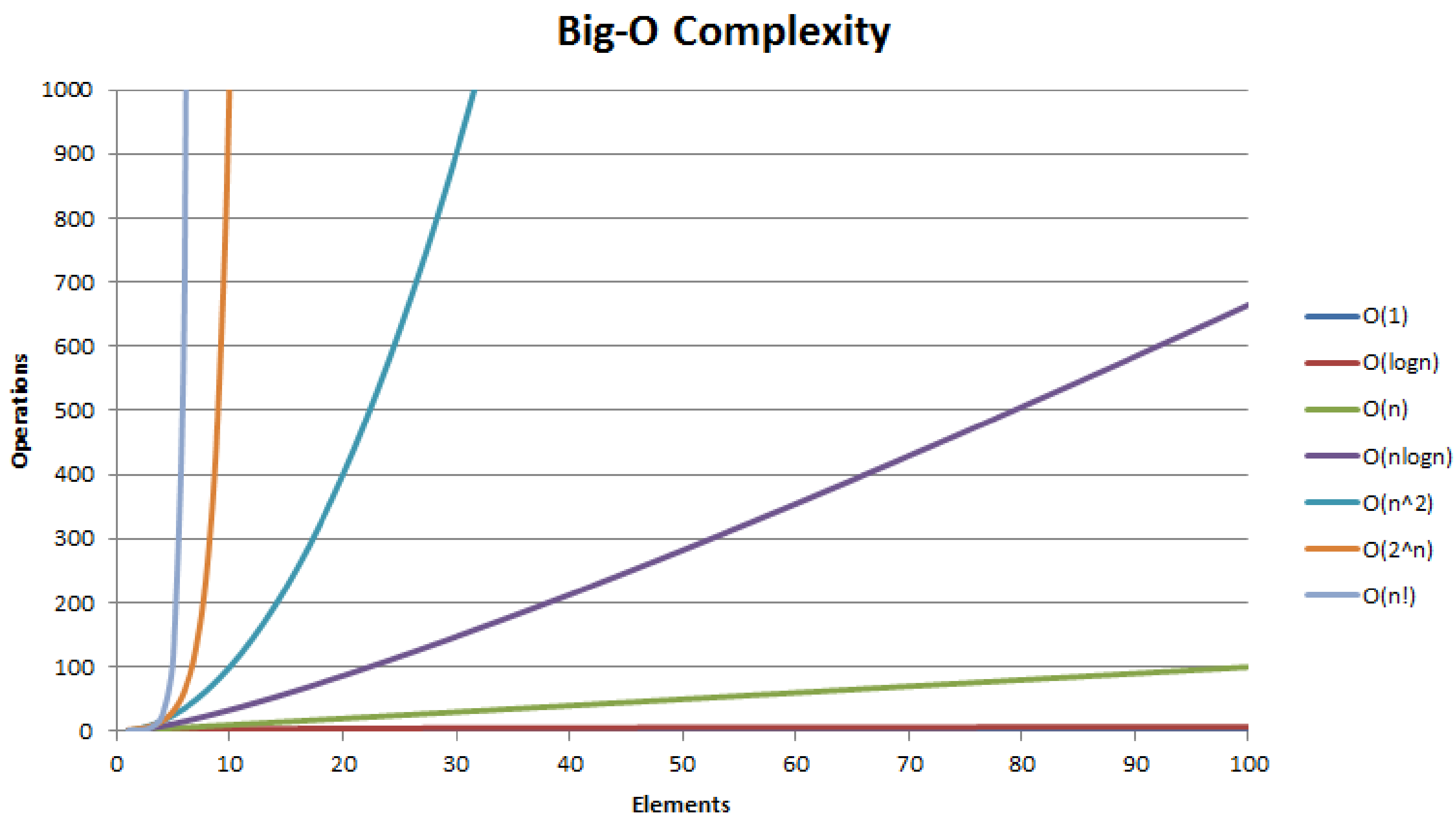
INTRODUCTION TO DATA STRUCTURES AND ALGORITHMS

DATA STRUCTURES AND ALGORITHMS

INTRODUCTION TO DATA STRUCUTRES AND ALGORITHMS

Content

- Concept of Algorithms and Data structures
- Algorithms
  - Complexity analyses
  - Big O notation
  - Calculation
  - Task with examples
    - Kadane's Algorithm
- Problem solving paradigm
- Data Structures
- Pointers and Memory
- Basics of templates and STL (C++ 11)



THE CONCEPT OF ALGORITHMS AND DATA STRUCTURES

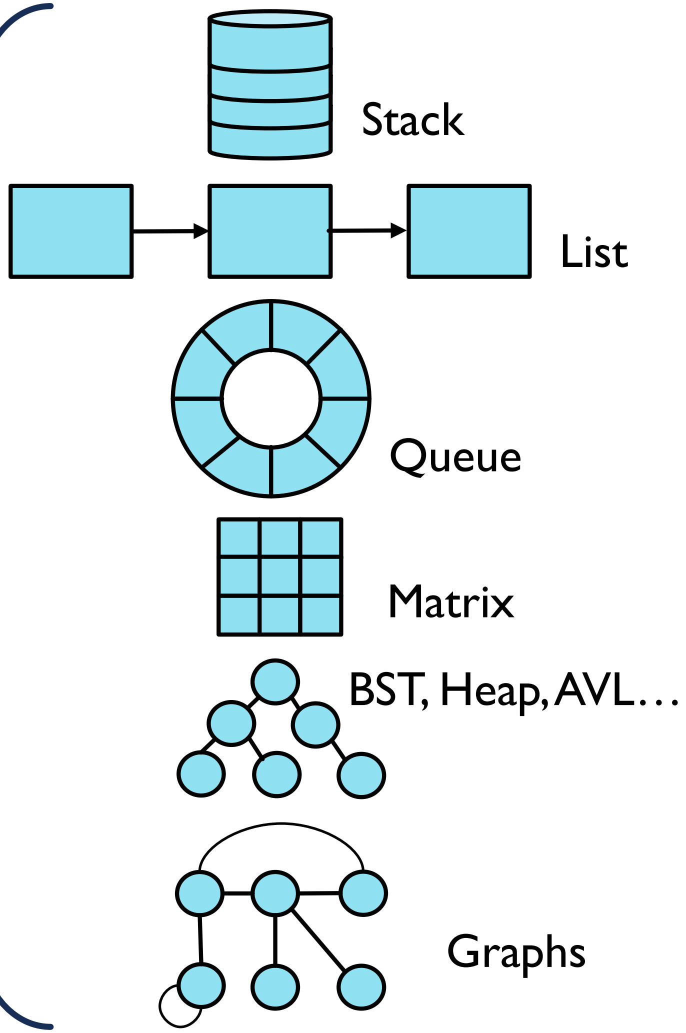
Algorithms

- sorting
- searching
- parsing
- traversing
- transforming
- Encoding & hashing

Processing

Data structures and Algorithms used to solve calculation tasks

Data structures



Software developing

Calculation tasks

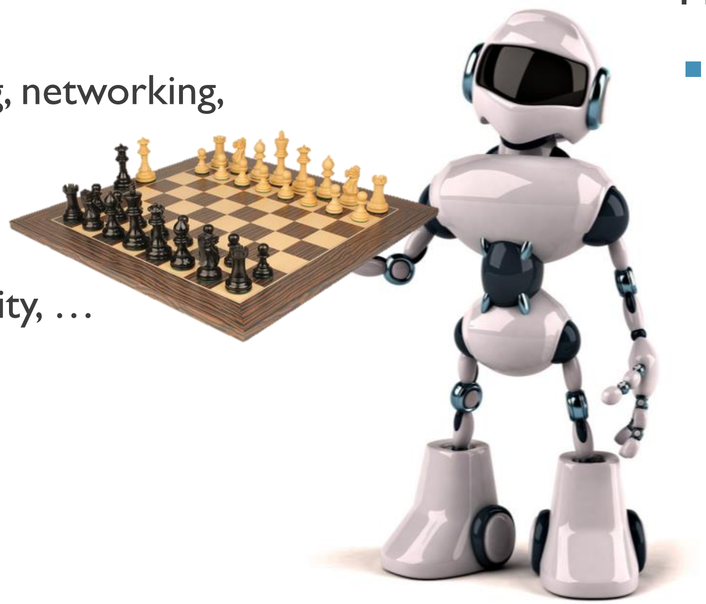
- Analyzing a task
- Select optimal data structure
- Select optimal algorithm
- compare with other approaches and complexity analysis

Not calculation task

- File system organizing
- Objects relations
- Design Patterns
- Refactoring
- Network
- Message passing
- Protocols and standards.
- Transactions and Databases
- Testing ……

WHY STUDY ALGORITHMS

- Internet
  - Web search, packet routing, distributed file sharing, ...
- Automation
  - Chain of process analysis, simulation, …
- Biology
  - Human genome project, protein folding, …
- Computers
  - Circuit layout, databases, caching, networking, compilers, …
- Computer graphics
  - Movies, video games, virtual reality, …
- Security
  - Cell phones, e-commerce, voting machines, …
- Multimedia
  - Cell phones, e-commerce, voting machines, …
- Social Networks
  - Recommendations, news feeds, advertisements, …
- Physics
  - N-body simulation, particle collision simulation, …



COMPLEXITY AND ASYMPTOTIC NOTATION

DATA STRUCTURES AND ALGORITHMS

RAM MODEL

The RAM (Random Access machine) model of computation

- Each simple operation (+, *, –, =, if, call) takes exactly one time step.

$$1 \begin{cases} A = B + C \\ X = A * 2 \\ Z = (X * C + A) * B \end{cases}$$

- Loops and subroutines are not considered simple operations. Instead, they are the composition of many single-step operations.

$$n \begin{cases} for(i = 0;\ i < n;\ i++) \\ \qquad A = B + C \end{cases}$$

$$n^2 \begin{cases} for(i = 0;\ i < n;\ i++) \\ \quad for(j = 0; j < n;\ j++) \\ \qquad A = B + C \end{cases}$$

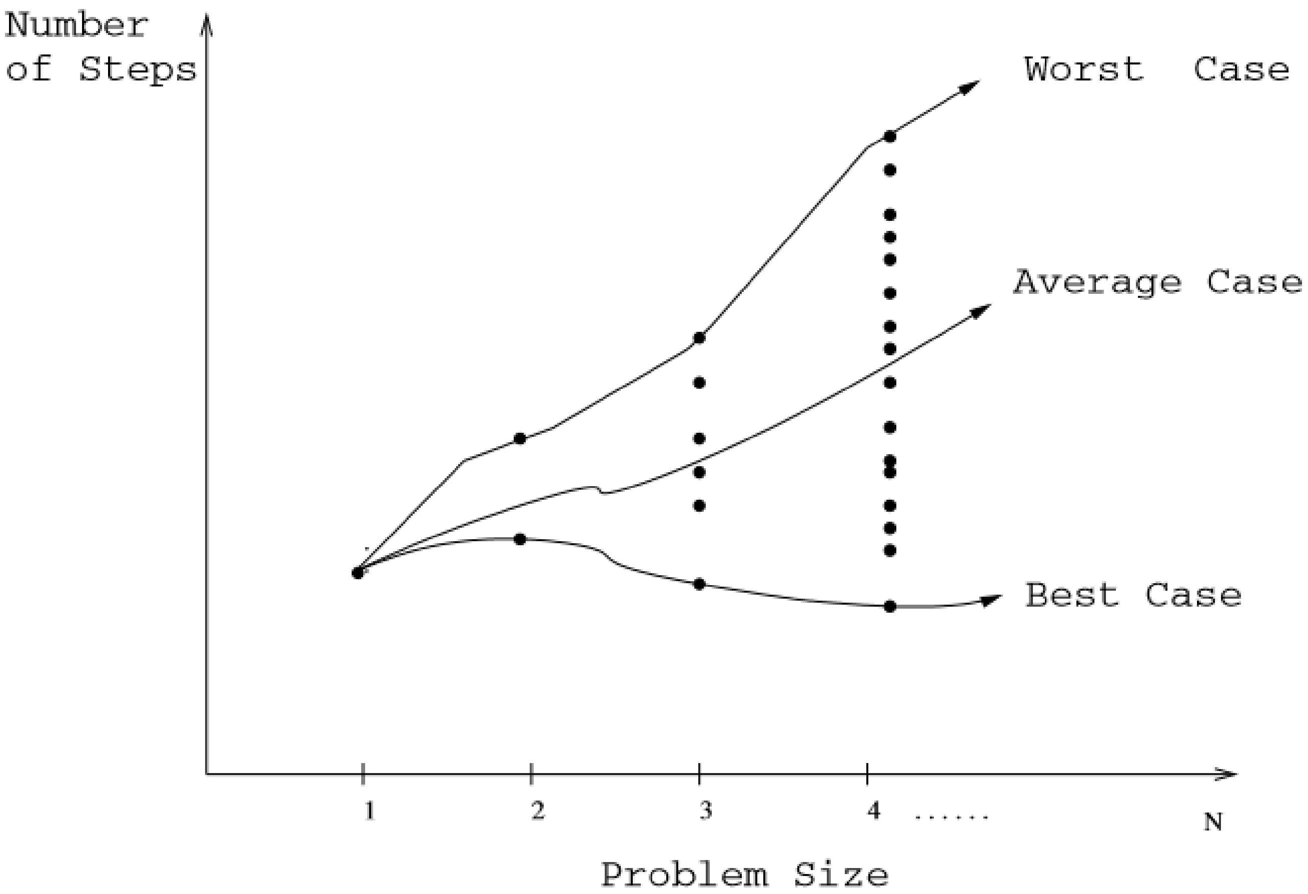
- Each memory access takes exactly one time step. Further, we have as much memory as we need. The RAM model takes no notice of whether an item is in cache or on the disk

Machine-independent algorithm design depends upon a hypothetical computer called the Random Access Machine or RAM. Under this model of computation, we are confronted with a computer where:

BEST, WORST, AND AVERAGE-CASE COMPLEXITY

Algorithms classification by speed

- The worst-case complexity of the algorithm is the function defined by the maximum number of steps taken in any instance of size n. This represents the curve passing through the highest point in each column
- The best-case complexity of the algorithm is the function defined by the minimum number of steps taken in any instance of size n. This represents the curve passing through the lowest point of each column.
- The average-case complexity of the algorithm, which is the function defined by the average number of steps over all instances of size n.

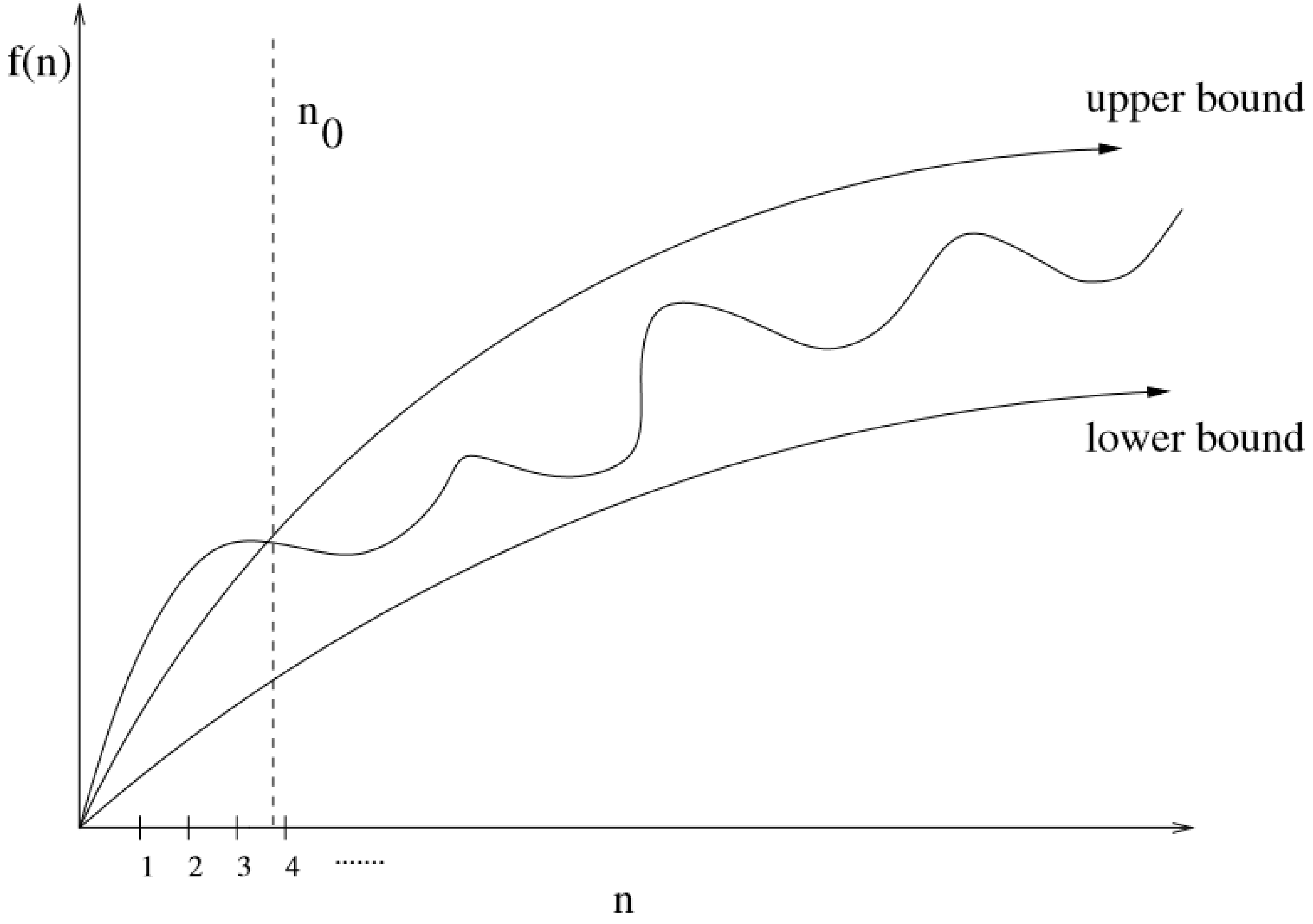


Using the RAM model of computation, we can count how many steps our algorithm takes on any given input instance by executing it

THE BIG O NOTATION

Formal Notation

- $f(n) = O(g(n))$ means $c \cdot g(n)$ is an upper bound on $f(n)$. Thus there exists some constant c such that $f(n)$ is always $\leq\ c \cdot g(n)$, for large enough n (i.e. , $n\ \geq\ n0$ for some constant $n0$).
- $f(n)\ =\ \Omega(\ g(n))$ means $c \cdot g(n)$ is a lower bound on $f(n)$. Thus there exists some constant c such that $f(n)$ is always $\geq\ c \cdot g(n)$, for all $n\ \geq\ n0$.
- $f(n)\ =\ \Theta(\ g(n))$ means $c1\ \cdot g(n)$ is an upper bound on $f(n)\ and c2\ \cdot g(n)$ is a lower bound on $f(n)$, for all $n\ \geq\ n0$. Thus there exist constants c1 and c2such that $f(n) \leq\ c1 \cdot g(n)\ and\ f(n) \geq\ c2 \cdot g(n)$. This means that $g(n)$ provides a nice, tight bound on $f(n)$.



Upper and lower bounds valid for n>n 0 smooth out the behavior of complex functions

WHY IT'S MATTER

The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing A million Hight-level instructions per second. In case, were the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1,000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10,000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100,000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1,000,000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

ANALYSIS OF ALGORITHMS

1) $O(1)$: Time complexity of a function (or set of statements) is considered as $O\ (1)$ if it doesn't contain loop, recursion and call to any other non-constant time function. A loop or recursion that runs a constant number of times is also considered as $O(1)$. For example the following loop is $O(1)$.

```
// Here c is a constant
   for (int i = 1; i <= c; i++) {
        // some O(1) expressions
   }
```

2) $O(n)$: Time Complexity of a loop is considered as $O(n)$ if the loop variables is incremented / decremented by a constant amount. For example following functions have $O(n)$ time complexity.

```
// Here c is a positive integer constant
for (int i = 1; i <= n; i += c) {
     // some O(1) expressions
}
```

```
for (int i = n; i > 0; i -= c) {
     // some O(1) expressions
}
```

ANALYSIS OF ALGORITHMS

3) $O(n^c)$: Time complexity of nested loops is equal to the number of times the innermost statement is executed. For example the following sample loops have $O(n^2)$ time complexity

```
for (int i = 1; i <=n; i += c) {
        for (int j = 1; j <=n; j += c) {
            // some O(1) expressions
        }
}
```

```
for (int i = n; i > 0; i += c) {
        for (int j = i+1; j <=n; j += c) {
            // some O(1) expressions
        }
}
```

4) $O(Logn)$ Time Complexity of a loop is considered as $O(Logn)$ if the loop variables is divided / multiplied by a constant amount.

```
for (int i = 1; i <=n; i *= c) {
        // some O(1) expressions
}
```

```
for (int i = n; i > 0; i /= c) {
        // some O(1) expressions
}
```

ANALYSIS OF ALGORITHMS

5) $O(LogLogn)$ Time Complexity of a loop is considered as $O(LogLogn)$ if the loop variables is reduced / increased exponentially by a constant amount.

```
// Here c is a constant greater than 1
for (int i = 2; i <=n; i = pow(i, c)) {
    // some O(1) expressions
}
```

```
//Here fun is sqrt or cuberoot or any other constant root
for (int i = n; i > 0; i = fun(i)) {
   // some O(1) expressions
}
```

COMBINE OF TIME COMPLEXITIES IN CONSECUTIVE LOOPS

When there are consecutive loops, we calculate time complexity as sum of time complexities of individual loops

```
for (int i = 1; i <=m; i += c) {
        // some O(1) expressions
}

for (int i = 1; i <=n; i += c) {
        // some O(1) expressions
}
```

Time complexity of above code is $O(m) + O(n)$ which is $O(m+n)$ If m == n, the time complexity becomes $O(2n)$ which is $O(n)$.

EXAMPLE OF ALGORITHM CALCULATION

INSERTION-SORT(A)	*cost*	*times*
for $j = 2$ **to** $A.length$	c_1	n
$key = A[j]$	c_2	$n-1$
// Insert $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$.	0	$n-1$
$i = j - 1$	c_4	$n-1$
while $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
$i = i - 1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
$A[i+1] = key$	c_8	$n-1$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

TASK: MAXIMUM SUBARRAY SUM

DATA STRUCTURES AND ALGORITHMS

TASK: MAXIMUM SUBARRAY SUM

Task

Given an array of n numbers, our task is to calculate the maximum subarray sum, i.e., the largest possible sum of a sequence of consecutive values in the array

Example

Input

-4	3	6	-4	7	2	-8	3

Data processing

Output

-4	3	6	-4	7	2	-8	3

$\underbrace{3 \quad 6 \quad -4 \quad 7 \quad 2}_{14}$

Result

Max sum is 14

MAXIMUM SUBARRAY SUM: ALGORITHM I

```
int best = 0;

for (int a = 0; a < n; a++) {
   for (int b = a; b < n; b++) {
      int sum = 0;
      for (int k = a; k <= b; k++) {
            sum += array[k];
      }
     best = max(best, sum);
    }
}
cout << best << "\n"
```

Algorithm I

Complexity $O(n^3)$

Too Slow ☹

MAXIMUM SUBARRAY SUM: ALGORITHM 2

```
int best = 0;

for(int a = 0; a < n; a++){

   int sum = 0;
   for (int b = a; b < n; b++){

      sum += array[b];
      best = max(best,sum);
  }
}
cout << best << "\n";
```

Algorithm 2

Complexity $O(n^2)$

Better than the first but could be faster 😐

MAXIMUM SUBARRAY SUM: ALGORITHM 3

```
int best = 0;
int sum = 0;

for (int k = 0; k < n; k++){
   sum = max(array[k],sum+array[k]);
   best = max(best,sum);
}

cout << best << "\n";
```

Algorithm 3

Complexity $O(n)$

Just one iteration ☺

Kadane's algorithm

SORTING ALGORITHMS

Name	Average	Worst	Memory	Stable
Bubble Sort	n^2	n^2	1	Yes
Selection Sort	n^2	n^2	1	No
Insertion Sort	n^2	n^2	1	Yes
Shell Sort	-	$n\log^2 n$	1	No
Binary Tree sort	$n \log n$	$n \log n$	n	Yes
Merge Sort	$n \log n$	$n \log n$	Depends	Yes
Heap Sort	$n \log n$	$n \log n$	1	No
Quick Sort	$n \log n$	n^2	$\log n$	Depends
Bucket Sort	$n k$	$n^2 k$	$n k$	Yes
Counting Sort	$n + k$	$n + k$	$n + 2^k$	Yes
LSD Radix Sort	$n k/d$	$n k/d$	n	Yes
MSD Radix sort	$n k/d$	$n k/d$	$n + k/d\ 2^d$	Yes

N: the number of items to be sorted
K: the size of each key
D: the digit size used by the implementation

PROBLEM SOLVING PARADIGM

DATA STRUCTURES AND ALGORITHMS

ALGORITHM DESIGN

Problem solving paradigm

- Brute Force
- Divide and conquer
- Greedy Algorithm
- Dynamic programming
- Backtracking

BRUTE FORCE

Idea

Construct an algorithm in a way to solve the task by checking all the possible combinations until finding the successful result

Easy to program, but too slow: $O(n^2), O(n^3)$ ...

Example

$$\begin{cases} x^2+y^2+z^2=C \\ x \times y \times z = B \\ x+y+z=A \end{cases}$$

$x, y, z \in [-100, 100]$

$x = ?\quad y = ?\quad z = ?$

Brute Force solution

```
bool sol = false;
int x, y, z;
for (x = -100; x <= 100; x++)
     for (y = -100; y <= 100; y++)
          for (z = -100; z <= 100; z++)
               if (y != x && z != x && z != y &&
                    x + y + z == A && x * y * z == B &&
                    x * x + y * y + z * z == C) {
                         if (!sol)
                              cout<< x << y << z << endl;
                         sol = true;
               }
```

Algorithms

- Linear search
- Bubble sort
- Selection sort
- Insertion sort
- Complete search

DIVIDE AND CONQUER

Idea

Divide and Conquer algorithm solves a problem using following three steps.

- *Divide:* Break the given problem into subproblems of same type.
- *Conquer:* Recursively solve these subproblems
- *Combine:* Appropriately combine the answers

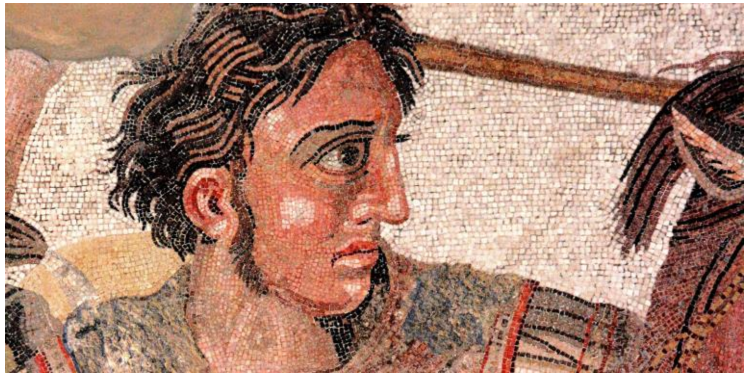
Example

Given an array of 1s and 0s which has all 1s first followed by all 0s. Find the number of 0s. Count the number of zeroes in the given array.

Divide and conquer solution

```
int firstZero(int arr[], int low, int high)
{
    if (high >= low) {
        // Check if mid element is first 0
        int mid = low + (high - low)/2;
        if (( mid == 0 || arr[mid-1] == 1) &&
              arr[mid] == 0)
            return mid;
        if (arr[mid] == 1)  // If mid element is not 0
            return firstZero(arr, (mid + 1), high);
        else  // If mid element is 0, but not first 0
            return firstZero(arr, low, (mid -1));
    }
    return -1;
}
int countZeroes(int arr[], int n)
{
    // Find index of first zero
    int first = firstZero(arr, 0, n-1);
    // If 0 is not present at all
    if (first == -1)
        return 0;

    return (n - first);
}
```



Algorithms

- Merge Sort
- Quick Sort
- Binary Search
- Karatsuba algorithm
- Segment tree's build and query

GREEDY

Idea

The algorithm makes the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem.

Apply the global rule (sort)

Get what you want on each step

Suppose the client needs to take 173 "manats" from ATM. The task is to write program for ATM to give the minimum amount of money

Greedy solution

```
int money[] = {20, 5, 10, 100, 1};
int amount = 173;
int div;

// apply the global rule for all items (sort)
sort(money, money+sizeof(money)/sizeof(int), greater<int>());

for (int i = 0; i < sizeof(money)/sizeof(int); i++)
{
        if(money[i] <= amount){
                div = amount/money[i];
                cout<<div<<" number of "<<money[i]<<" manat"<<endl;
                amount -= div * money[i];
        }
}
```

Algorithms

- Kruskal's algorithm
- Prim's algorithm
- Egyptian fraction problem

DYNAMIC PROGRAMMING

Idea

Dynamic Programming is an algorithmic paradigm that solves a given complex problem by breaking it into subproblems and stores the results of subproblems to avoid computing the same results again.

```
int fib(int n)
{
  if (n == 0 )
      return 0;
  if(n == 1)
      return 1;
return fib(n-1) + fid(n-2);
}
```

Recursive solution creates a tree in stack where each node has to be calculated (even if was)

```
int fib(int n)
{
  vector<int> f(n+1);
  int i;
  f[0] = 0;
  f[1] = 1;
  for (i = 2; i <= n; i++)
      f[i] = f[i-1] + f[i-2];

  return f[n];
}
```

Algorithms

- Dijkstra Algorithm
- Kadane's algorithm
- Floyd Warshall's algorithm

BACKTRACKING

Idea

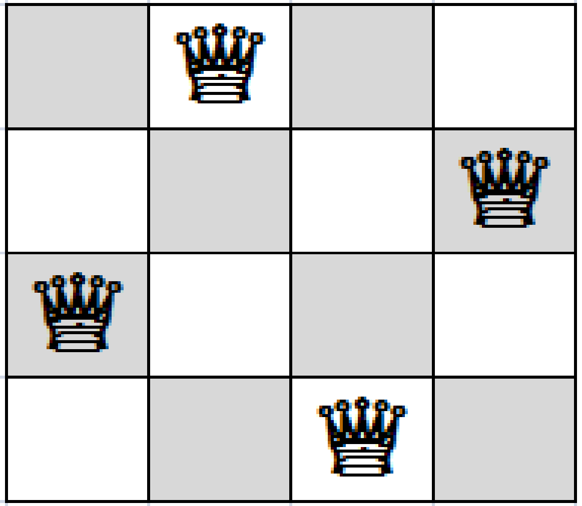
Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions, and abandons each partial candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.

```
bool isSafe(int board[N][N], int row, int col){
    int i, j;
    for (i = 0; i < col; i++){
        if (board[row][i])
            return false;
    }
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--){
        if (board[i][j])
            return false;
    }
    for (i = row, j = col; j >= 0 && i < N; i++, j--){
        if (board[i][j])
            return false;
    }
    return true;
}
```

Brute Force VS Backtracking

- Brute Force: you generate all the possible combinations you can and then you check if any of them is the answer you want
- Backtracking : In each step, you check if this step satisfies all the conditions If it does : you continue generating subsequent solutions
  If not : you go one step backward to check for another path

```
bool solveNQUtil(int board[N][N], int col)
{
    if (col >= N)
        return true;
    for (int i = 0; i < N; i++)
    {
        if ( isSafe(board, i, col) )
        {
            /* Place this queen in board[i][col] */
            board[i][col] = 1;
            /* recur to place rest of the queens */
            if ( solveNQUtil(board, col + 1)
                board[i][col] = 0; // BACKTRACK
        }
    }
return false;
}
```



Algorithms

- Hamiltonian cycle
- The Knight’s tour problem
- Rat in maze

POINTERS AND STRUCTURES

CREATING THE STRUCTURES

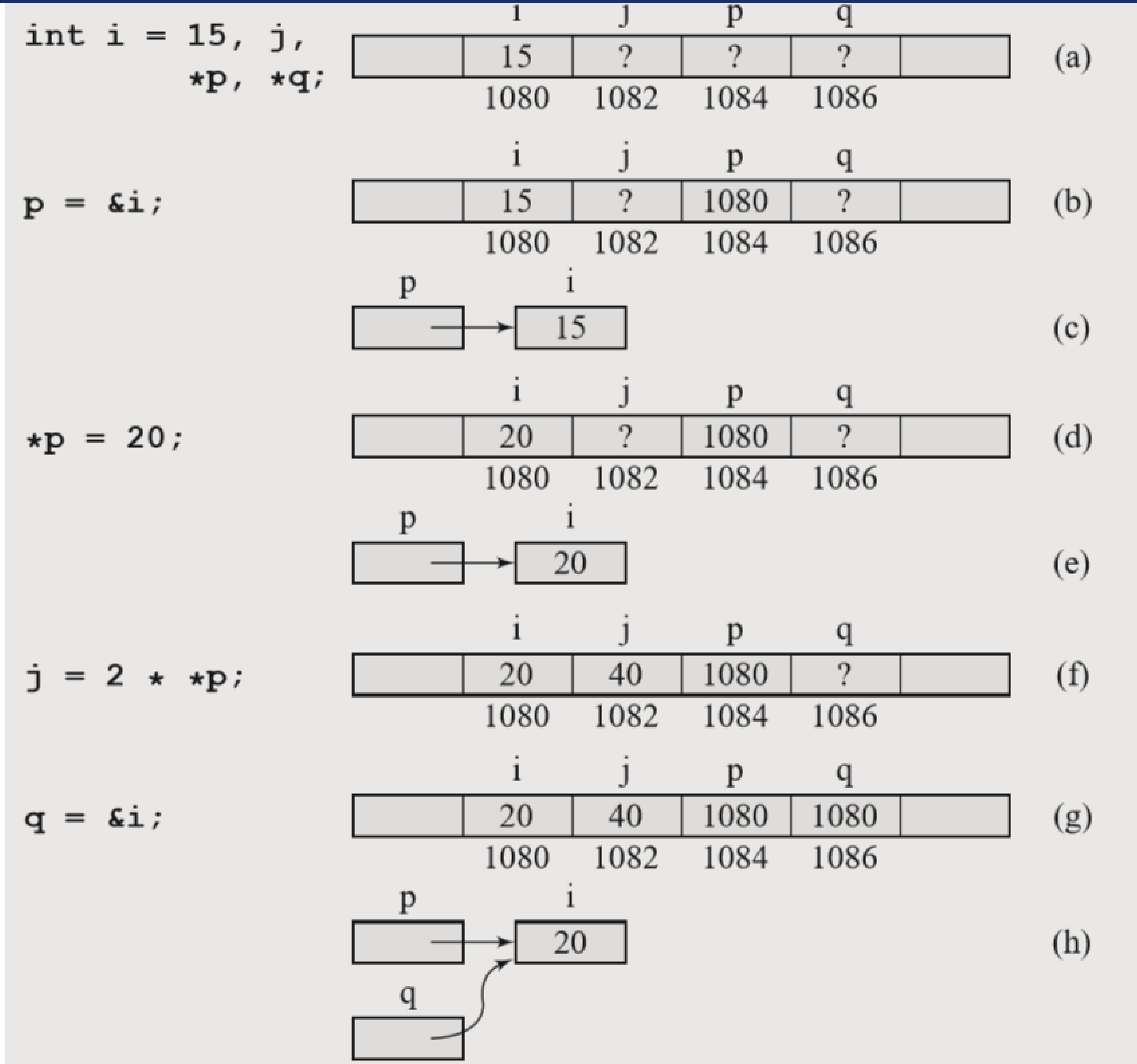
POINTERS

```
p = new int;

delete p;

p = new int;
```

- Variable that stores an address of other variable
- Variable used in dynamic memory allocation



POINTERS AND ARRAYS

- The declarations specify that a is a pointer to a block of memory that can hold five integers. The pointer a is fixed; that is, a should be treated as a constant so that any attempt to assign a value to a, as in **a = p;** or in **a++;** is considered a compilation error. Because a is a pointer, pointer notation can be used to access cells of the array a. For example, an array notation used in the loop that adds all the numbers in a

```
int a[5], *p;
```

```
for (sum = a[0], i = 1; i < 5; i++)
     sum += a[i];
```

```
for (sum = *a, i = 1; i < 5; i++)
     sum += *(a + i);
```

```
for (sum = *a, p = a+1; p < a+5; p++)
     sum += *p;
```

TEMPLATES AND STL

DATA STRUCTURES AND ALGORITHMS

TEMPLATE

Definition

- Templates are a mechanism for classes and functions in C++ to have type parameters. This is a concept similar to generics in Java 1.5. In this course, we won't bother much with the details of how to create templatized classes and functions; we will simply use the ones provided by the C++ Standard Template Library (STL).

```
int min(int a, int b){
    return a < b ? a: b;
}

template< class C >
C min( C a, C b ) {
     return a < b ? a : b;
}
```

```
int x = 2, y = 3, z = min( x, y );
double d = 0.5, e = 1,
f = min( d, e );
string s = "the first s", t = "the second string", u = min( s, t );
```

STL BY TEMPLATE EXAMPLE

- It is possible to define classes with template parameters, too. For example, the STL defines a class pair that simply stores 2 things.

```
template< class A, class B >
class pair {

     public:
        A first;
        B second;
        pair( A a, B b ) {
            first = a; second = b;
        }
};
```

- pair has 2 member variables (or fields): first is of type A and second is of type B. To create a pair, you need to manually specify what A and B are, like this.

```
pair< string, double > p( "pi", 3.14 );
p.second = 3.14159;
```

STL VECTOR

```
#include<vector>
```

- The STL vector class is designed to provide all the functionalities of normal arrays, but with several extra features that make vector easier to use and handle. Like an array, a vector holds a collection of elements of the same data type, in a contiguous sequence for O(1) random access. The elements of a vector are accessed using the square bracket operator "[]". You can set and retrieve these elements just like you would in a normal array. Example 1 above shows how a vector is used.

```
vector< int > v( 10 );
   for( int j = 0; j < 10; j++ ) v[j] = j * j;
```

This will create a vector of 10 integers (all garbage initially) and fill it up with the first 10 perfect squares (0, 1, 4, 9, ..., 81). Note that the square brackets operator is overloaded for vectors, so `v` can be used just like an array. Now we will go into more details about using vector and set.

VECTOR EXAMPLES

```
vector< int > v( 10 );
 for( int j = 0; j < 10; j++ )
     v[j] = j * j;   // v is of size 10, filled with squares
     v.resize( 20 ); // After resizing to size 20, the
                     //first 10 elements
                     // remain unchanged. The rest are undefined.
  for (int j = 10; j < 20; j++ )
     v[j] = j * j; // v is now size 20, filled with squares
```

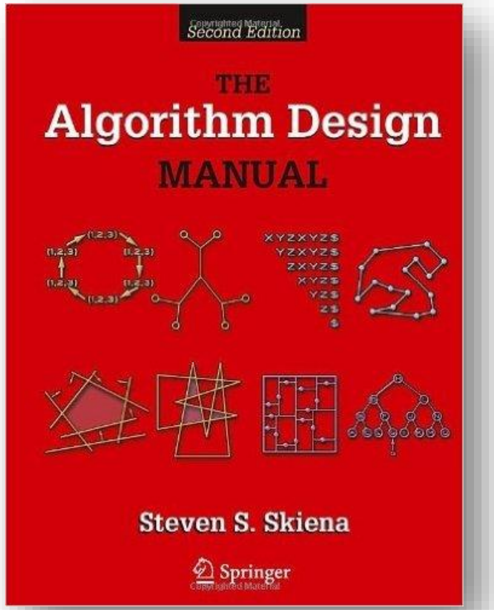
```
vector< int > v; // Creates an empty vector
for( int j = 0; j < 10; j++ )
v.push_back( j * j ); // append one element to v
/**
* At this point, the vector v has automatically grown to size 10.
* You can add more elements to v using more push_back calls.
**/
```

QUEUE

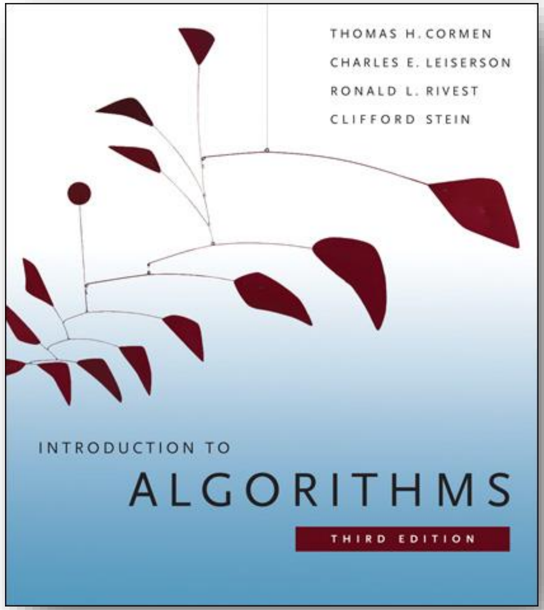
- queue is a First-In-First-Out (FIFO) container, whereas a stack is Last-In-First-Out (LIFO). A deque is both a stack and a queue. The following demonstrates just about all you can do with a queue:

```
queue< int > Q;                     // Construct an empty queue
   for ( int i = 0; i < 3; i++ )
          Q.push( i );              // Pushes i to the end of the queue
                                    // Q is now { "0", "1", "2" }
   int sz = Q.size();               // Size of queue is 3
   while( !Q.empty() ) {            // Print until Q is empty
       int element = Q.front();     // Retrieve the front of the queue
       Q.pop();                     // REMEMBER to remove the element!
       cout << element << endl;     // Prints queue line by line
   }
```

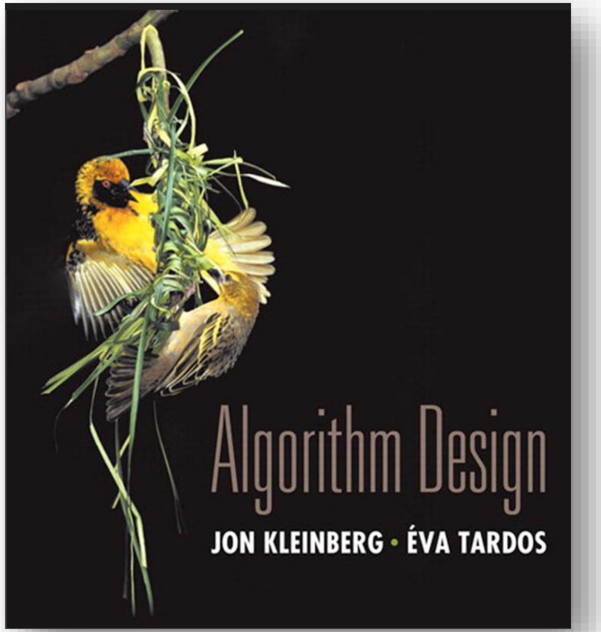
LITERATURE



Stieven Skienna
Algorithms design manual
Chapter 2.1 RAM model of computation
Page 33.



Thomas H. Cormen
Introduction to Algorithms
Chapter 2.2: Analyzing algorithms
Page 23.



Jon Kleinberg, Eva Tardos
Chapter 2: Basics of Algorithm Analysis
Page 30