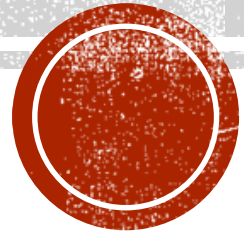
ANALYSIS OF ALGORITHMS



Ekim 2013

DATA TYPE

A data type is

a collection of objects and

a set of operations that act on these objects

int

{0, +1, -1, +2, -2, ...INT_MAX, INT_MIN}

INT_MAX, INT_MIN (limits.h)

+, -, *, %, /, atoi,...

ABSTRACT DATA TYPE

- A data type.
- The distinction between spefication and implementation.
  package (Ada), class (C++, Java)
- Implementation independent.

The specification

the names of the every function

the type of its arguments

the type of its results

what the function does (without implementation details)

ADT- NATURAL NUMBERS

structure Natural_Number is

objects : an ordered subrange of the integers starting at zero and ending at the maximum integer (INT_MAX) on the computer.

functions :

for all x, y ∈ Nat_Number, TRUE, FALSE ∈ Boolean

+, -, <, == are the usual integer operations

Nat_No Zero()	::=	0
Boolean IsZero(x)	::=	**if**(x) **return** FALSE
		else return TRUE
Nat_No Ad(x,y)	::=	...
Boolean Equal(x,y)	::=	...
Nat_No Successor(x)	::=	...
Nat_No Substract(x, y)	::=	...

A GOOD PROGRAM

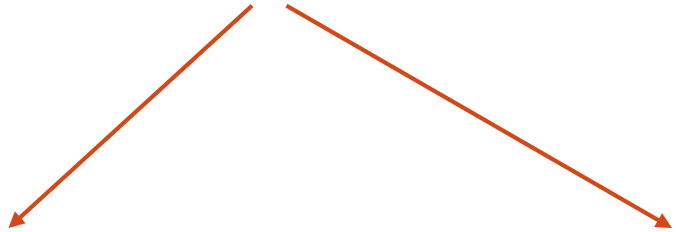
- Meet requirements
- Run correctly
- Be easy to read and understand
- Be easy to debug
- Be easy to modify
- Be documented

- Run efficiently

  does the program efficiently use primary and secondary storage

  is the program's running time acceptable for the task

PERFORMANCE EVALUATION



performance analysis

machine independent

performance measurement

machine dependent

to identify inefficient code segments

SPACE COMPLEXITY

The amount of memory that the program needs to run to completion.

1. Fixed Space Requirements

   - not depend on the size of the program's input and output
   - the instruction space (to store the code)
   - the space for simple variabes, fixed-size structured variables, and constants.

2. Variable Space Requirements

SPACE COMPLEXITY

2. Variable Space Requirements

   - The space needed by structured variables whose size depends on the particular instance I. $S_p(I)$
   - The additional space required when a function uses recursion.

$$S(P) = c + S_p(I)$$

SPACE COMPLEXITY - EXP

```
float abc(float a, float b, float c)
{
      return a+b+b*c + (a+b-c) / (a+b) + 4.00;
}
```

This function has only fixed space requirements.

$S_{abc}(I) = 0$

SPACE COMPLEXITY - EXP

```
float sum(float list[], int n)
{
      float tempSum = 0;
      int i;
      for( i = 0; i <n; i++)
            tempSum += list[i];
      return tempSum;
}
```

This function has variable space requirements depend on the array passed to the function.

$S_{sum}(I) = n$ (passing the entire array)

$S_{sum}(I) = 0$ (passing the address of the first element)

SPACE COMPLEXITY - EXP

```
float rsum(float list[], int n)
{
      if(n) return rsum(list, n-1) + list[n-1];
      return 0;
}
```

The space requirement for one recursive call :

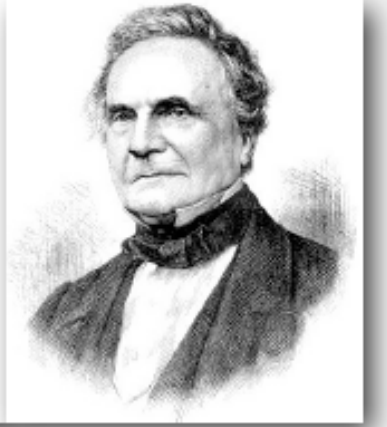
SIZE =

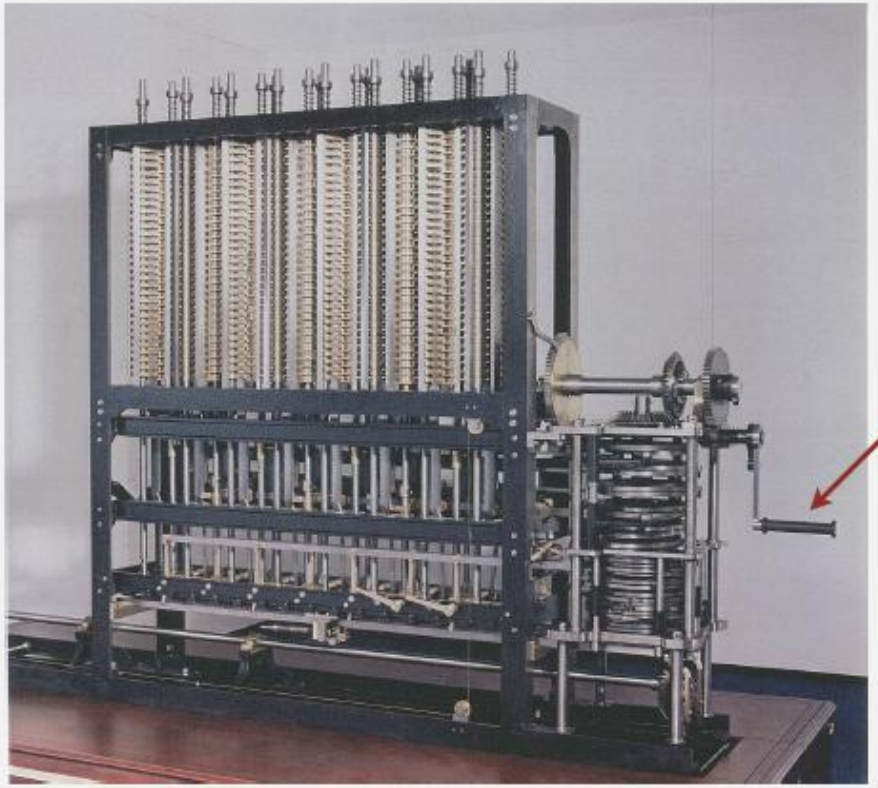
the space required for the two parameters +

the return address

n * SIZE

" As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise—By what course of calculation can these results be arrived at by the machine in the shortest time? " — *Charles Babbage (1864)*





how many times do you have to turn the crank?

Analytic Engine

TIME COMPLEXITY

The amount of computer time that the program needs to run to completion.

T(P) = compile time + execution time

Total Running Time : sum of cost x frequency for all operations ***by Knuth***

$$T_p(n) = c_a ADD(n) + c_s SUB(n) + C_l LDA(n) + c_{st} STA(n)$$

- Depends on machine
- Need to analyse the program to determine set of operations
- Frequency depend on algorithm and input data

RUNNING TIME

System Independent Effects

Algorithm

Input Data

System Dependent Effects

Hardware: CPU, memory, cache, …

Software : compiler, interpreter, garbage collector, …

System : operating system, network, other apps, …

slide by R. Sedgewick and K. Wayne

TIME COMPLEXITY - EXP

```
float sum(float list[], int n)
{
        float tempSum = 0;
        int i;
        for( i = 0; i <n; i++)
                tempSum += list[i];
        return tempSum;
}
```

Steps

1

n+1
n
1

TIME COMPLEXITY - EXP

```
float sum(float list[], int n)
{
        float tempSum = 0;
        int i;
        for( i = 0; i <n; i++)
                tempSum += list[i];
        return tempSum;
}
```

```
count++ (for assignment)

{ count++ (for the for loop)
count++ (for assignment) }
count++ (the last exec. of for)
count++ (for return)
```

2n + 3 steps.

TIME COMPLEXITY - EXP

```
float rsum(float list[], int n)                    Steps
{
      if(n)                                        n+1
            return rsum(list, n-1) + list[n-1];    n
      return 0;                                    1
}
```

2n + 2 steps.

TIME COMPLEXITY - EXP

```
void add(int a[][MAX_SIZE], int b[][MAX_SIZE], int c[][MAX_SIZE],
                    int rows, int cols)
{
      int i, j;
      for (i = 0; i < row; i++)
            for(j = 0; j < cols; j++)
                  c[i][j] = a[i][j] + b[i][j];
}
```

2rows*cols + 2rows + 1 steps.

TYPES OF ANALYSES

Best case : Lower bound on cost.

Determined by "easiest" input.

Provides a good for all inputs.

Worst case : Upper bound on cost.

Determined by "most difficult" input..

Provides a guarantee for all inputs.

Average case : Expected cost for random input.

Need a model for "random" input.

Provides a way to predict performance.

slide by R. Sedgewick and K. Wayne

TYPES OF ANALYSES - EXP

Ex 1. Array accesses for brute-force 3-SUM.

Best: $\sim \frac{1}{2} N^3$

Average: $\sim \frac{1}{2} N^3$

Worst: $\sim \frac{1}{2} N^3$

Ex 2. Compares for binary search.

Best: ~ 1

Average: $\sim \lg N$

Worst: $\sim \lg N$

ASYMPTOTIC NOTATION – BIG OH : O

f(n) = O(g(n)) iff there exist positive constants c and n_0 such that

f(n) <= cg(n) for all n, n >= n_0

$3n+2 = O(n)$	as $3n+2 <= 4n$	for all n>= 2
$10n^2+4n+2 = O(n^2)$	as $10n^2 + 4n + 3 <= 11n^2$	for n>=5
$6*2^n + n^2 = O(2^n)$	$6*2^n + n^2 <= 7*2^n$	for n>=4

$3n+3 = O(n)$	Correct, OK.
$3n+3 = O(n^2)$	Correct, NO!

ASYMPTOTIC NOTATION – OMEGA : Ω

f(n) = O(g(n)) iff there exist positive constants c and n_0 such that

f(n) >= cg(n) for all n, n >= n_0

$3n+2 = \Omega(n)$	as 3n+2 >= 3n	for all n>= 1
$10n^2+4n+2 = \Omega(n^2)$	as $10n^2 + 4n + 2 >= n^2$	for n>=1
$6*2^n + n^2 = \Omega(2^n)$	$6*2^n + n^2 >= 2^n$	for n>=1

$3n+3 = \Omega(n)$	Correct, OK.
$3n+3 = \Omega(1)$	Correct, NO!

ASYMPTOTIC NOTATION – THETA : Θ

$f(n) = O(g(n))$ iff there exist positive constants c_1, c_2, and n_0 such that

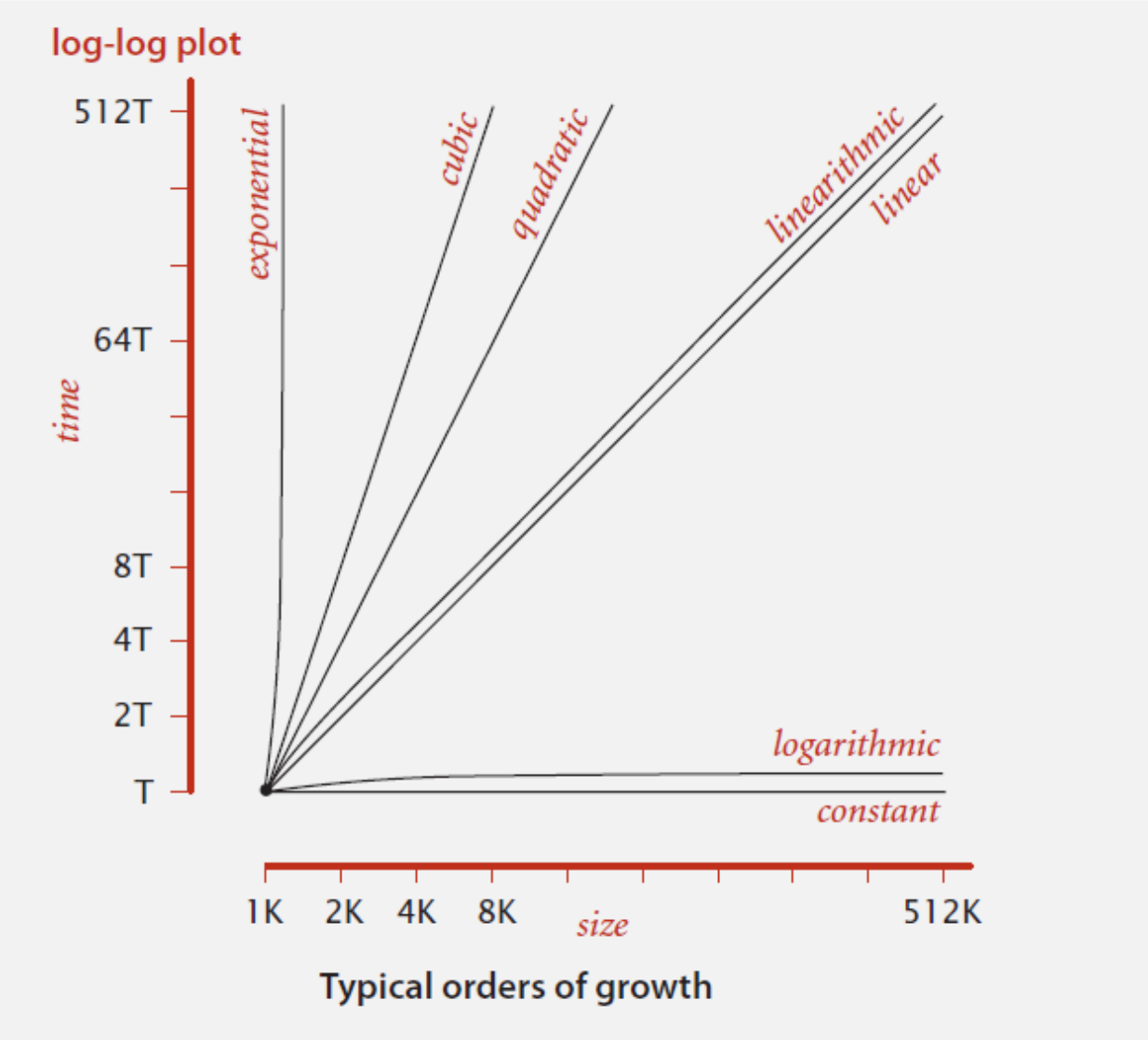
$c_1 g(n) <= f(n) <= c_2 g(n)$ for all n, $n >= n_0$

$3n+2 = \Theta(n)$	as $3n+2 >= 3n$	for all $n>= 1$
$10n^2+4n+2 = \Theta(n^2)$	as $10n^2 + 4n + 2 >= n^2$	for $n>=1$
$6*2^n + n^2 = \Theta(2^n)$	$6*2^n + n^2 >= 2^n$	for $n>=1$

$3n+3 = \Omega(n)$ Correct, OK.

$3n+3 = \Omega(2n)$ Correct, not used!

notation	provides	example	shorthand for	used to
Big Theta	asymptotic order of growth	$\Theta(N^2)$	$\frac{1}{2} N^2$ $10 N^2$ $5 N^2 + 22 N \log N + 3N$ ⋮	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	$10 N^2$ $100 N$ $22 N \log N + 3 N$ ⋮	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	$\frac{1}{2} N^2$ N^5 $N^3 + 22 N \log N + 3 N$ ⋮	develop lower bounds



Typical orders of growth

1

logN

N

NlogN

N^2

N^3

2^N

slide by R. Sedgewick and K. Wayne

order of growth	name	typical code framework	description	example	T(2N) / T(N)
1	constant	`a = b + c;`	statement	add two numbers	1
log N	logarithmic	`while (N > 1) { N = N / 2; ... }`	divide in half	binary search	~ 1
N	linear	`for (int i = 0; i < N; i++) { ... }`	loop	find the maximum	2
N log N	linearithmic	[see mergesort lecture]	divide and conquer	mergesort	~ 2
N^2	quadratic	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) { ... }`	double loop	check all pairs	4
N^3	cubic	`for (int i = 0; i < N; i++) for (int j = 0; j < N; j++) for (int k = 0; k < N; k++) { ... }`	triple loop	check all triples	8
2^N	exponential	[see combinatorial search lecture]	exhaustive search	check all subsets	T(N)

growth rate	problem size solvable in minutes			
	1970s	1980s	1990s	2000s
1	any	any	any	any
log N	any	any	any	any
N	millions	tens of millions	hundreds of millions	billions
N log N	hundreds of thousands	millions	millions	hundreds of millions
N^2	hundreds	thousand	thousands	tens of thousands
N^3	hundred	hundreds	thousand	thousands
2^N	20	20s	20s	30

MAGIC SQUARE

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

```
#define MAX_SIZE 15
void main()                                          //based on Coxeter's rule
{
        int square[MAX_SIZE][MAX_SIZE];
        int i, j, row, column;
        int count, size;

        //Kare boyutunu kullanıcıdan al, kontrol et
        for(i= 0; i < size; i++)
                for(j = 0; j < size; j++)
                        square[i][j] = 0;

        square[0][(size-1)/2] = 1;
        i = 0;
        j = (size-1)/2;

        for(count = 2; count < = size*size; count++){
                row = (i-1< 0) ? (size-1) : (i-1);
                column = (j-1 < 0) ? (size-1) : (j -1);

                if(square[row][column])          //down
                        i = (++i) % size;
                else{                                    //square is unoccupied
                        i = row;
                        j = column;
                }
                square[i][j] = count;
        }
}
```