Algorithm Analysis

04.10.2023

Problem Solving

Problem Solving: Main Steps

1. Problem definition (model the problem)
2. Algorithm design (find an algorithm that solves the modeled problem)
3. Algorithm analysis (fast enough? fits into the memory?)
4. Implementation
5. Testing
6. *Maintenance*

1. Problem Definition

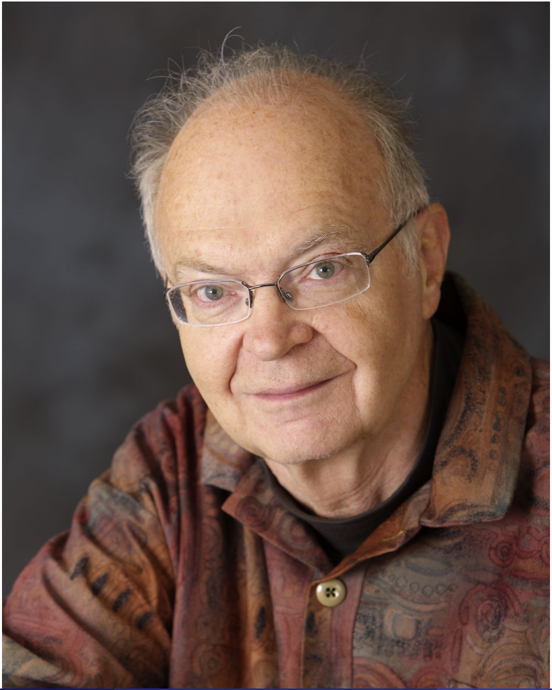
- Try to understand what are the main elements of the problem that need to be solved. (Functional requirements)
  - Calculate the mean of "n" numbers.
  - Scan a document in English and translate it to Turkish.
- What are the performance requirements? (Non functional requirements)

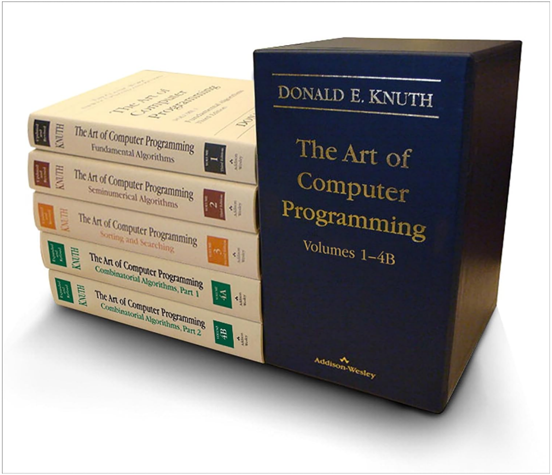
Functional vs Non-Functional Requirements

Functional requirements: “Any Requirement Which Specifies What The System Should Do.”

Non-Functional requirements: "Any Requirement That Specifies How The System Performs A Certain Function"

Non-functional requirements can be considered as quality attributes.




DONALD E. KNUTH
The Art of Computer Programming
Volumes 1–4B
Addison-Wesley
The Art of Computer Programming
Fundamental Algorithms
The Art of Computer Programming
Seminumerical Algorithms
The Art of Computer Programming
Sorting and Searching
The Art of Computer Programming
Combinatorial Algorithms, Part 1
The Art of Computer Programming
Combinatorial Algorithms, Part 2

2. Algorithm Design

- Algorithm: A clearly specified set of simple instructions to be followed to solve a problem
- An algorithm can be described in natural language, pseudo-code, diagrams etc.
- Knuth's characterization (five properties as requirements for an algorithm):
  - Input: Zero or more quantities (externally produced)
  - Output: One or more quantities
  - Definiteness: Clarity, precision of each instruction
  - Finiteness: The algorithm has to stop after a finite (may be very large) number of steps
  - Effectiveness: Each instruction has to be basic enough and feasible

3. Algorithm Analysis

- Given an algorithm, will it satisfy the requirements?
- Given a number of algorithms to perform the same computation, which one is "best"?
- The analysis required to estimate the "resource use" of an algorithm
  - Space complexity
    - How much space is required ?
  - Time complexity
    - How much time does it take to run the algorithm ?
- Often, we have to deal with estimates!

4,5,6: Implementation, Testing, Maintenance

- Implementation
  - Decide on the programming language
  - Write clean, well documented code
- Test (How long?)
- Maintenance
  - Bug fixing, version management, new features etc.

Problem Solving: Life-cycle

1. Problem definition
2. Algorithm design
3. Algorithm analysis
4. Implementation
5. Testing
6. *Maintenance*

Rationale

-> Why do we need Algorithm Analysis in our profession?

-> What should be Analysed?

3. Algorithm Analysis

- Space complexity
  - How much space is required ?
- Time complexity
  - How much time does it take to run the algorithm ?

Space Complexity

- Space complexity = The amount of memory required by an algorithm to run to completion
- Some algorithms may be more efficient if data completely loaded into memory
  - Need to look also at system limitations e.g. Classify 20GB of text in various categories – Can I afford to load the entire collection?

Space Complexity (cont'd)

1. Fixed part: The size required to store certain data/variables, that is independent of the size of the problem:
   - e.g. name of the data collection
   - Same size for classifying 2GB or 1MB of texts
2. Variable part: Space needed by variables, whose size is dependent on the size of the problem:
   - e.g. actual text
   - Load 2GB of text vs. load 1MB of text

The space requirement $S(P)$ of any program P may therefore be written as $S(P) = c + S_p$ where c is a constant and S_p is the instance characteristics which depends on a particular instance of a problem.

Space Complexity - Example

```
int sum = 0;
int n;
int* numbers;
// Read "n" from user
numbers = new int[n];
// Read "n" integers from user
// Calculate the sum
```

```
int sum = 0;
int n;
// Read "n" from user
int current_number;
// Read "n" many times current_number from user
// Calculate the sum
```

Time Complexity

The analysis required to estimate the resource use of an algorithm is generally a theoretical issue, and therefore a formal framework is required. Some definitions;

$T(N)$

Represents the running time -in terms of the number of basic operations- of an algorithm for input size N.

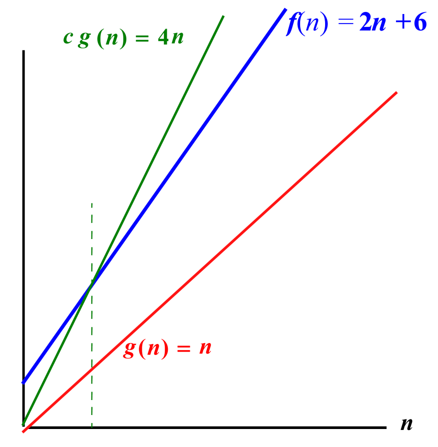
- $T(N) = O(f(N))$ if there are positive constants c and n_0 such that $T(N) \leq cf(N)$ when $N \geq n_0$.
- $T(N) = \Omega(g(N))$ if there are positive constants c and n_0 such that $T(N) \geq cg(N)$ when $N \geq n_0$.
- $T(N) = \Theta(h(N))$ if and only if $T(N) = O(h(N))$ and $T(N) = \Omega(h(N))$
- $T(N) = o(p(N))$ if, for all positive constants c, there exists n_0 such that $T(N) < cp(N)$

Big-Oh Notation

- Why do we need these definitions?
  - We try to establish a relative order among functions.
  - We want to compare their **relative rate of growth**
- $1000N$ vs N^2
- We can say $1000N = O(N^2)$, which is know as Big-Oh notation. Instead of saying "order ..." we say "Big-Oh ...."

Graphic Illustration

- $f(n) = 2n + 6$
- We need to find a function $g(n)$ and a constant c and n_0 such as $f(n) < cg(n)$ when $n > n_0$



- $g(n) = n$ and $c = 4$ and $n_0 = 3 \rightarrow f(n)$ is $O(n)$
- The order of $f(n)$ is n

Big Omega, Big Theta

- $O(f(n))$: Big-Oh – $f(N)$ is an upper bound on $T(N)$
- $\Omega(f(n))$: Big Omega – $f(N)$ is a lower bound on $T(N)$
- $\Theta(f(n))$: Big Theta – $f(N)$ is a tight bound on $T(N)$
- Consider the difference:
  - $3n + 3$ is $O(n)$ and is $\Theta(n)$
  - $3n + 3$ is $O(n^2)$ but is not $\Theta(n^2)$ (Why? -Technical Answer)

Little-Oh

- $o(f(n))$: Little-oh – $f(n)$ is $o(g(n))$ if $f(n)$ is $O(g(n))$ and $f(n)$ is not $\Theta(g(n))$
  - $2n + 3$ - Is it $o(n^2)$? (Why? -Technical Answer)
  - $2n + 3$ - Is it $o(n)$? (Why? -Technical Answer)

In other words...

- "$f(n)$ is $O(g(n))$" $\rightarrow$ growth rate of $f(n) \leq$ growth rate of $g(n)$
- "$f(n)$ is $\Omega(g(n))$" $\rightarrow$ growth rate of $f(n) \geq$ growth rate of $g(n)$
- "$f(n)$ is $\Theta(g(n))$" $\rightarrow$ growth rate of $f(n) =$ growth rate of $g(n)$
- "$f(n)$ is $o(g(n))$" $\rightarrow$ growth rate of $f(n) <$ growth rate of $g(n)$

Examples

- N^3 grows faster than N^2, we can say $N^2 = O(N^3)$ or $N^3 = \Omega(N^2)$.
- $f(N) = N^2$ and $g(N) = 2N^2$ grows at same rate, we can say $f(N) = O(g(N))$ and $f(N) = \Omega(g(N))$, hence we can say $f(N) = \Theta(g(N))$
- What about $f(n) = 4n^2$? Is it $O(n)$?
  - Find a c and n_0 such that $4n^2 < cn$ for any $n > n_0$

Rules

- If $f_1(n)$ is $O(g_1(n))$ and $f_2(n)$ is $O(g_2(n))$
  - $f_1(n) + f_2(n)$ is $O(g_1(n) + g_2(n)) = O(max(g_1(n), g_2(n)))$
  - $f_1(n) \times f_2(n)$ is $O(g_1(n) \times g_2(n))$
- If $f(N)$ is a polynomial of degree k, then $f(N) = \Theta(N^k)$.
- $log^k N$ is $O(N)$ for any constant k
- The relative growth rate of two functions can always be determined by computing their limit (But using this method is almost always an overkill)

$$\lim_{n\to\infty} f_1(n)/f_2(n)$$

Big-Oh and Growth Rate

- We can use the big-Oh notation to rank functions according to their growth rate

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows faster	Yes	No
$f(n)$ grows faster	No	Yes
Same growth	Yes	Yes

Typical Grow Rates

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Some Numbers

log n	n	n log n	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

Big-Oh Rules

- If $f(n)$ is a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions!
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$ (which is also true)"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

More examples

- $50n^3 + 20n + 4$ is $O(n^3)$
  - Would be correct to say $f(n)$ is $O(n^3 + n)$?
    - Not useful, as n^3 exceeds by far n, for large values
  - Would be correct to say $f(n)$ is $O(n^5)$?
    - OK, but $g(n)$ should be as close as possible to $f(n)$
- $3log(n) + n(log(n)) = O(?)$

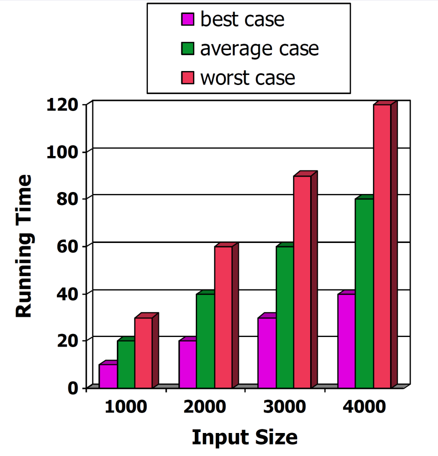
Simple Rule: Drop lower order terms and constant factors

Inappropriate Expressions

$f(n) \cancel{\leq} O(g(n))$
$f(n) \cancel{\geq} O(g(n))$

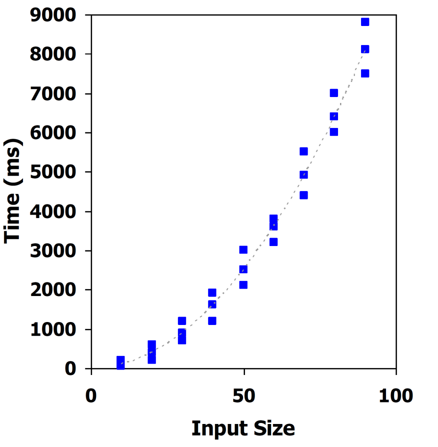
Analyzing Running Time

- The running time of an algorithm varies with the inputs, and typically grows with the size of the inputs.
- To evaluate an algorithm or to compare two algorithms, we focus on their relative rates of growth wrt the increase of the input size.
- Generally the average running time is difficult to determine theoretically, hence we focus on the worst case running time. Which one to focus depends on the particular problem.



Experimental Approach

- Write a program to implement the algorithm.
- Run this program with inputs of varying size and composition.
- Get an accurate measure of the actual running time (e.g. system call date).
- Plot the results.
- Problems?



Maximum Subsequence Sum Problem

Given a set of integers $A_1, A_2, \ldots, A_N$, find the maximum value of

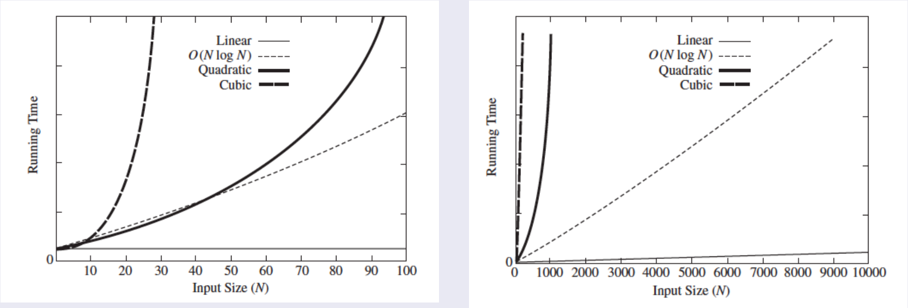
$$\sum_{k=i}^{j} A_k$$

For convenience, the maximum subsequence sum is zero if all the integers are negative.

Running Times in Seconds For Different Algorithms

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

Running Times vs N



Limitations of Experimental Studies

- The algorithm has to be implemented, which may take a long time and could be very difficult.
- Results may not be indicative for the running time on other inputs that are not (or could not since they do not finish in reasonable time) included in the experiments.
- In order to compare two algorithms, the same hardware and software must be used.

Use a Theoretical Approach

- Based on the high-level description of the algorithms, rather than language dependent implementations
- An evaluation of the algorithms that is independent of the hardware and software environments is possible
  $\rightarrow$ **Generality**

Pseudocode

- High-level description of an algorithm.
- More structured than plain English.
- Less detailed than a program.
- Preferred notation for describing algorithms.
- Hides program design issues.

Example: find the maximum element of an array

Algorithm 1 arrayMax(A, n)

```
Input array A of n integers
Output maximum element of A
currentMax ← A[0].
for i ← 1 to n − 1 do
    if A[i] > currentMax then
        currentMax ← A[i]
return currentMax
```

Pseudocode

- Control flow
  - if ... then ... [else ...]
  - while ... do ...
  - repeat ... until ...
  - for ... do ...
  - Indentation replaces braces
- Method declaration
  Algorithm method (arg [, arg...])
  Input ...
  Output
- Method call
  - var.method (arg [, arg. . .])
- MethodReturn value
  - return expression
- MethodExpressions
  - $\leftarrow$Assignment (equivalent to =)
  - = Equality testing (equivalent to ==)
  - n^2 Superscripts and other mathematical formatting allowed

Primitive Operations

- The basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important
- Use comments
- Instructions have to be basic enough and feasible

- Examples:
  - Evaluating an expression
  - Assigning a value to a variable
  - Calling a method
  - Returning from a method

Low Level Algorithm Analysis

- Based on primitive operations (low-level computations independent from the programming language)
- For example:
  - Make an addition = 1 operation
  - Calling a method or returning from a method = 1 operation
  - Index in an array = 1 operation
  - Comparison = 1 operation, etc.
- Method: Inspect the pseudo-code and count the number of primitive operations executed by the algorithm

Counting Primitive Operations

- By inspecting the code, we can determine the number of primitive operations executed by an algorithm, as a function of the input size.

Algorithm 2 arrayMax(A, n)	#operations
$currentMax \leftarrow A[0]$.	2
for $i \leftarrow 1$ *to* $n - 1$ **do**	2+n (init & loopEnd)
if A[i] $>$ currentMax **then**	2(n-1)
currentMax $\leftarrow$ A[i]	2(n-1)
{increment of counter i }	2(n-1) (inc & loopStart)
return currentMax	1
	Total 7n-1

Estimating Running Time

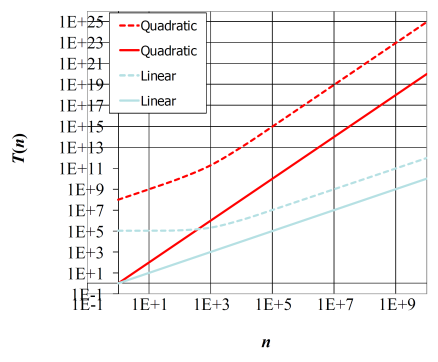
- Algorithm *arrayMax* executes $7n - 1$ primitive operations.
- Let's define
  $a :=$ Time taken by the fastest primitive operation
  $b :=$ Time taken by the slowest primitive operation
- Let $T(n)$ be the actual running time of *arrayMax*. We have
  $$a(7n - 1) \leq T(n) \leq b(7n - 1)$$
- Therefore, the running time $T(n)$ is bounded by two linear functions

Growth Rate of Running Time

- Changing computer hardware / software
  - Affects $T(n)$ by a constant factor
  - Does not alter the growth rate of $T(n)$
- The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm *arrayMax*

Constant Factors

- The growth rate is not affected by
  - Constant factors or
  - Lower-order terms
- Examples
  - $10^2 n + 10^5$ is a linear function
  - $10^5 n^2 + 10^8 n$ is a quadratic function



Growth Rates

- Growth rates of functions:
  - Linear $\approx n$
  - Quadratic $\approx n^2$
  - Cubic $\approx n^3$
- In a log-log chart, the slope of the line corresponds to the growth rate of the function

Algorithm Analysis

- We know:
  - Experimental approach – problems
  - Low level analysis – counting operations is not feasible
- Abstract even further
- Characterize an algorithm as a function of the "problem size"
- e.g.
  - Input data = array $\rightarrow$ problem size is N (length of array)
  - Input data = matrix $\rightarrow$ problem size is N $\times$ M

- We can utilize Big-Oh, Big Omega, Big Theta and Little-Oh notations to analyze running time of algorithms.
- These notations invented by Paul Bachmann, Edmund Landau, and others, collectively are called Bachmann-Landau notation or asymptotic notation.

Running Time Calculations - General Rules

- FOR loop
  - The number of iterations times the running time of the statements inside the loop.
- Nested loops
  - The product of the sizes of all the loops times the running time of the statement inside the loop.
  - Analyze inside out.
- Consecutive Statements
  - The sum of running time of each segment.
- If/Else
  - The testing time plus the larger of the running time of the cases.

Some Examples

```
for (i=0; i<n; i++)
   for (j=0; j<n; j++)
       k++;
```

$O(n^2)$

```
for (i=0; i<n; i++)
    k++;
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        k++;
```

$O(n^2)$

Some Examples

```
if (x > 0)
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            k++;
else
    for (i=0; i<n; i++)
        k++;
```

$O(n^2)$

Maximum Subsequence Sum Problem

Given a set of integers $A_1, A_2, \ldots, A_N$, find the maximum value of

$$\sum_{k=i}^{j} A_k$$

For convenience, the maximum subsequence sum is zero if all the integers are negative.

The First Algorithm

```
int MaxSubSum1(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;
}
```

$O(n^3)$

The Second Algorithm

```
int MaxSubSum2(vector<int> a) {
    int maxSum = 0;

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

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

$O(n^2)$

The Third Algorithm

```
// Recursive maximum contiguous sum algorithm
int maxSubSum3(vector<int> a) {
    return maxSumRec(a, 0, a.size() - 1);
}

// Used by driver function above, this one is called
    recursively
int maxSumRec(const vector<int>& a, int left, int right) {
    if (left == right) // base case
        if (a[left] > 0)
            return a[left];
        else
            return 0;

    int center = (left + right) / 2;
    int maxLeftSum = maxSumRec(a, left, center); //
        recursive call
    int maxRightSum = maxSumRec(a, center + 1, right); //
        recursive call

    int maxLeftBorderSum = 0, leftBorderSum = 0;
```

$O(n)$

$T(n') = T(n/2)$

The Third Algorithm (Cont.)

```
    for (int i = center; i >= left; --i) {
        leftBorderSum += a[i];
        if (leftBorderSum > maxLeftBorderSum)
            maxLeftBorderSum = leftBorderSum;
    }

    int maxRightBorderSum = 0, rightBorderSum = 0;
    for (int j = center+1; j <= right; ++j) {
        rightBorderSum += a[j];
        if (rightBorderSum > maxRightBorderSum)
            maxRightBorderSum = rightBorderSum;
    }

    return max3(maxLeftSum, maxRightSum,
        maxLeftBorderSum + maxRightBorderSum);
}

int max3(int x, int y, int z) {
    int max = x;
    if (y > max)
        max = y;
    if (z > max)
        max = z;
    return max;
}
```

Recurrence Relation

- $T(1) = 1$
- $T(n) = 2T(n/2) + O(n)$
  - $T(2) = ?$
  - $T(4) = ?$
  - $T(8) = ?$
  - $\ldots$
- If $n = 2^k$, $T(n) = n \times (k + 1)$
  - $k = log\ n$
  - $T(n) = n(log\ n + 1)$

Solving recursive equations by repeated substitution

$$
\begin{aligned}
T(n) &= T(n/2) + c \text{ substitute for } T(n/2) \\
&= T(n/4) + c + c \text{ substitute for } T(n/4) \\
&= T(n/8) + c + c + c \\
&= T(n/2^3) + 3c \text{ in more compact form} \\
&= \ldots \\
&= T(n/2^k) + kc \text{ "inductive leap"} \\
T(n) &= T(n/2^{log\ n}) + clogn \text{ "choose k = logn"} \\
&= T(n/n) + clogn \\
&= T(1) + clogn = b + clogn = \Theta(log\ n)
\end{aligned}
$$

Solving recursive equations by telescoping

$T(n) = T(n/2) + c$ initial equation
$T(n/2) = T(n/4) + c$ so this holds
$T(n/4) = T(n/8) + c$ and this ...
$T(n/8) = T(n/16) + c$ and this ...
...
$T(4) = T(2) + c$ eventually ...
$T(2) = T(1) + c$ and this ...
$T(n) = T(1) + c\log n$ sum equations, canceling the terms appearing on both sides
$T(n) = \Theta(\log n)$

Logarithms in the Running Time

- An algorithm is $O(log\ N)$ if it takes constant time to cut the problem size by a fraction (usually $\frac{1}{2}$).
- On the other hand, if constant time is required to merely reduce the problem by a constant amount (such as to make the problem smaller by 1), then the algorithm is $O(N)$
- Examples of the $O(log\ N)$
  - Binary Search
  - Euclid's algorithm for computing the greatest common divisor

The Fourth Algorithm

```
int MaxSubSum4(vector <int> a) {
    int maxSum=0, thisSum=0;

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

        if (thisSum>maxSum)
            maxSum=thisSum;
        else if (thisSum<0)
            thisSum=0;
    }
    return maxSum;
}
```

$O(n)$