Algorithm Analysis

October 12, 2016

Problem Solving: Main Steps

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

1. Problem Definition

- What is the task to be accomplished?
  - Calculate the average of the grades for a given student
  - Understand the talks given out by politicians and translate them in Chinese
- What are the time / space / speed / performance requirements ?

2. Algorithm Design / Specifications

- Algorithm: Finite set of instructions that, if followed, accomplishes a particular task.
- Describe: in natural language / pseudo-code / diagrams / etc.
- Criteria to follow:
  - 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
    - ~~Understand speech~~
    - ~~Translate to Chinese~~

Computer Algorithms+

- An algorithm is a procedure (a finite set of well-defined instructions) for accomplishing some tasks which,
  - given an initial state
  - terminate in a defined end-state
- The computational complexity and efficient implementation of the algorithm are important in computing, and this depends on using suitable data structures.

4,5,6: Implementation, Testing, Maintenance

- Implementation
  - Decide on the programming language to use
    - C, C++, Lisp, Java, Perl, Prolog, assembly, etc.
  - Write clean, well documented code
- Test, test, test
- Integrate feedback from users, fix bugs, ensure compatibility across different versions $\rightarrow$ Maintenance

3. Algorithm Analysis

- Space complexity
  - How much space is required
- Time complexity
  - How much time does it take to run the algorithm
- Often, we deal with estimates!

Space Complexity

- Space complexity = The amount of memory required by an algorithm to run to completion
  - Core dumps = the most often encountered cause is "dangling pointers"
- 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 [politics, tourism, sport, natural disasters, etc.] – 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

Space Complexity (cont'd)

- $S(P) = c + S(\textit{instance characteristics})$
  - c = constant
- Example:

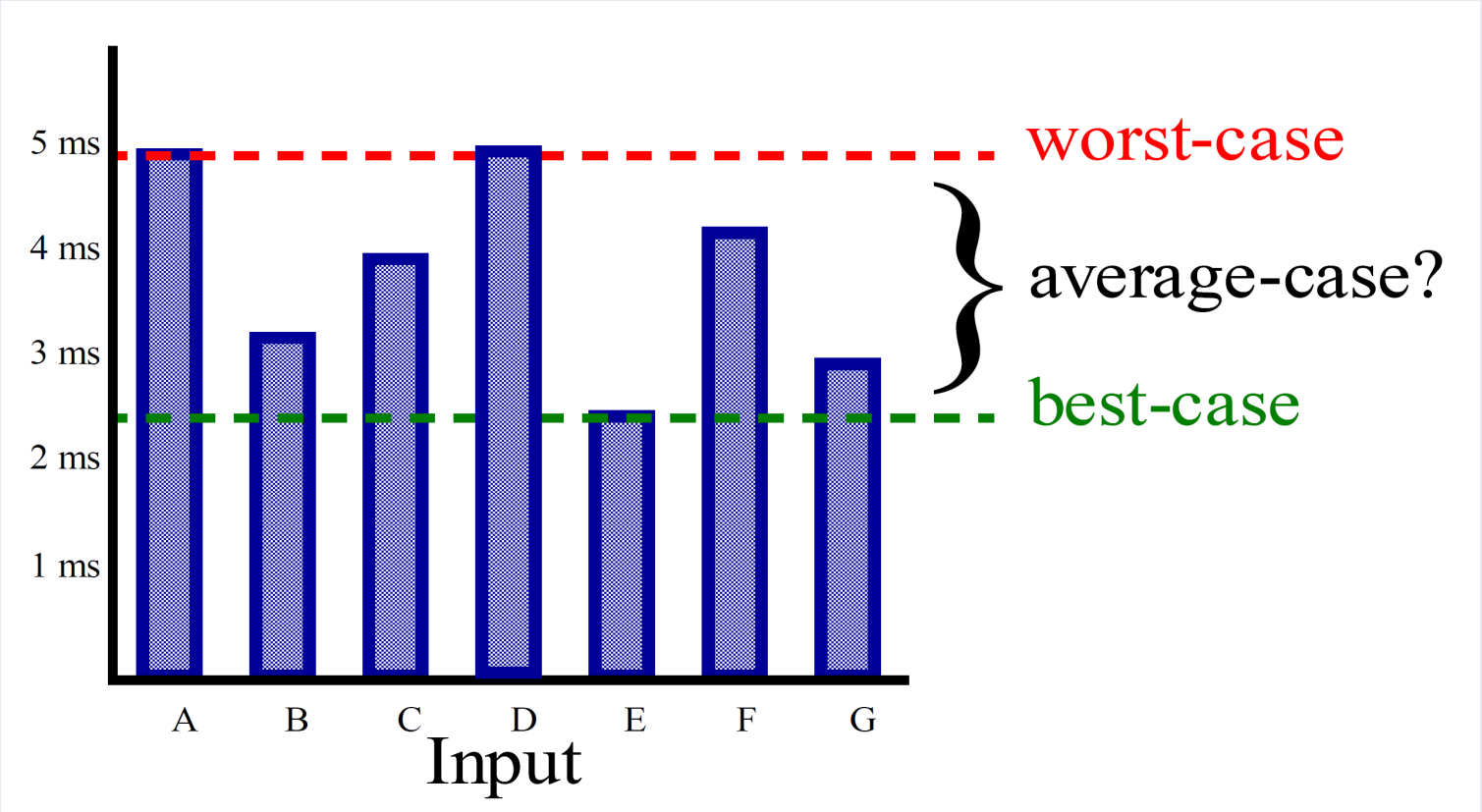
```
float summation(const float (&a) [10], int n)
{
        float s=0;

        for (int i = 0; i < n; i++ )
        {
                s+=a[i];
        }
        return s;
}
```

Space? one for `n`, one for `a` (passed by reference!), one for `s`, one for `i` $\rightarrow$ constant space!

Time Complexity

- Often more important than space complexity
  - space available (for computer programs!) tends to be larger and larger
  - time is still a problem for all of us
- 3-4GHz processors on the market
  - still ...
  - researchers estimate that the computation of various transformations for 1 single DNA chain for one single protein on 1 TerraHZ computer would take about 1 year to run to completion
- Algorithm's running time is an important issue.

Running time

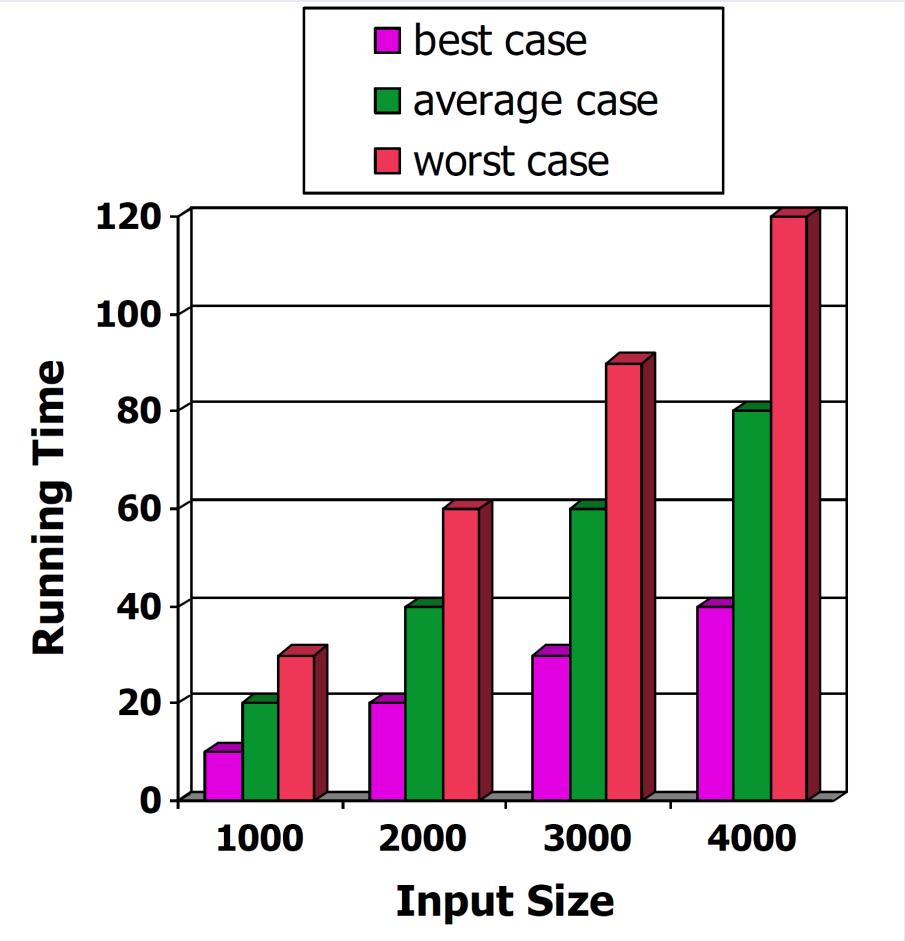


Suppose the program includes an `if-then` statement that may execute or not: $\rightarrow$ variable running time.

Typically algorithms are measured by their worst case

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.
- The average running time is difficult to determine.
- We focus on the worst case running time
  - Easier to analyze
  - Crucial to applications such as finance, robotics, and games



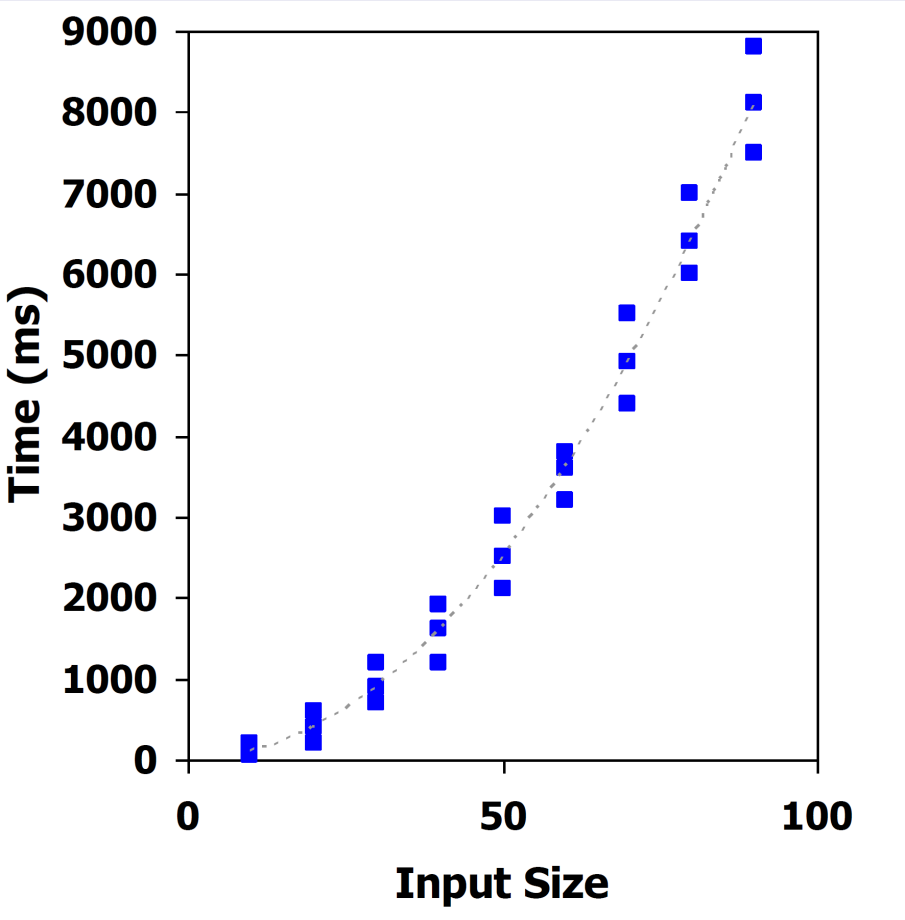
Running Time

- Problem: prefix averages
  - Given an array `X`
  - Compute the array `A` such that `A[i]` is the average of elements `X[0] ... X[i]`, for `i=0..n-1`
- Sol 1
  - At each step `i`, compute the element `X[i]` by traversing the array `A` and determining the sum of its elements, respectively the average
- Sol 2
  - At each step `i` update a sum of the elements in the array `A`
  - Compute the element `X[i]` as `sum/i`

Big question: Which solution to choose?

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?



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 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
- Makes possible an evaluation of the algorithms that is independent of the hardware and software environments
  $\rightarrow$ **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)

```
1: Input array A of n integers
2: Output maximum element of A
3: currentMax ← A[0].
4: for i ← 1 to n − 1 do
5:     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
1: *currentMax* $\leftarrow$ *A*[0].	2
2: **for** *i* $\leftarrow$ 1 *to n* $-$ 1 **do**	2+n
3: **if** A[i] $>$ currentMax **then**	2(n-1)
currentMax $\leftarrow$ A[i]	2(n-1)
4: {increment counter i }	2(n-1)
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

$$a(7n - 1) \leq T(n) \leq b(7n - 1)$$

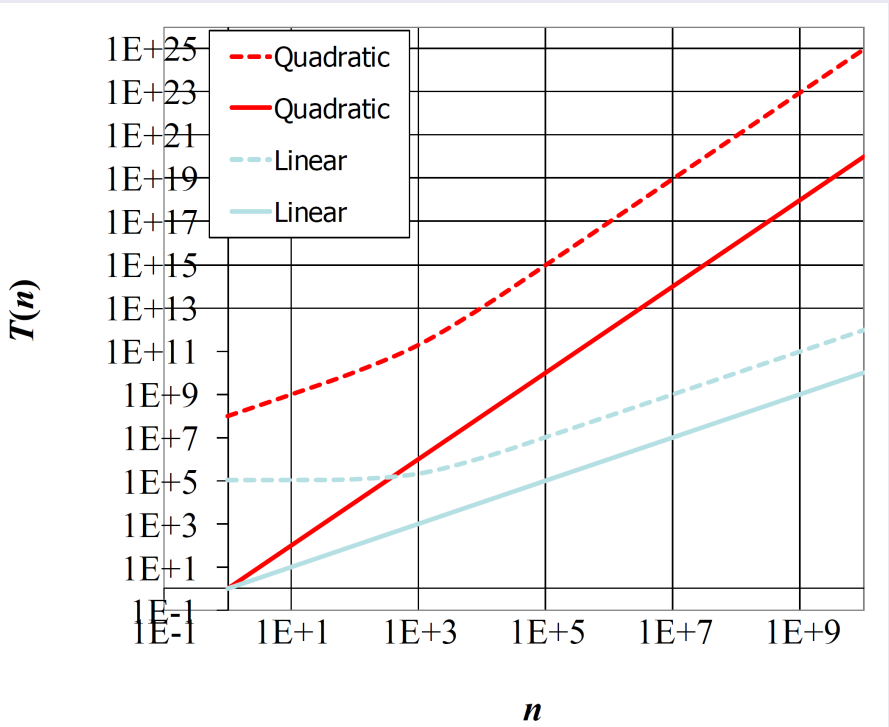
- Therefore, the running time $T(n)$ is bounded by two linear functions

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*

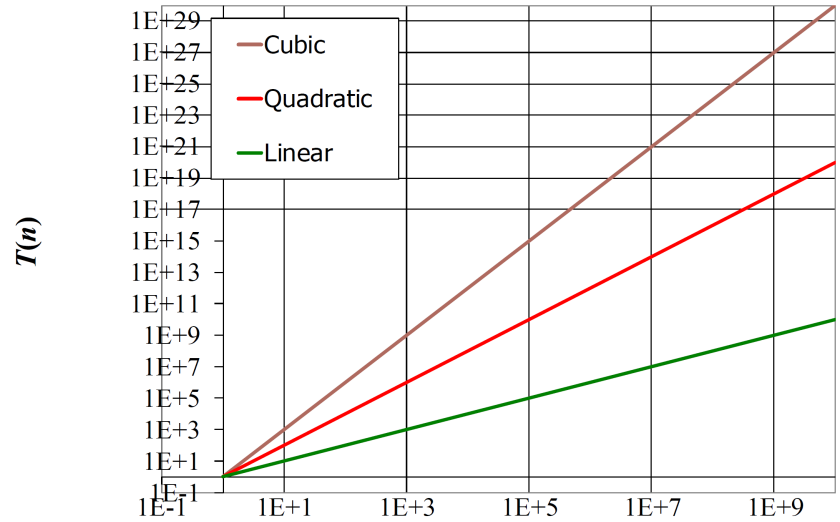
Constant Factors

- 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



Asymptotic Notation

- Need to abstract further
- Give an "idea" of how the algorithm performs
- n steps vs. $n + 5$ steps
- n steps vs. n^2 steps

Problem

- Fibonacci numbers
  - F[0] = 0
  - F[1] = 1
  - F[i] = F[i-1] + F[i-2] for i $\geq$ 2
- Pseudo-code
- Number of operations

Algorithm Analysis

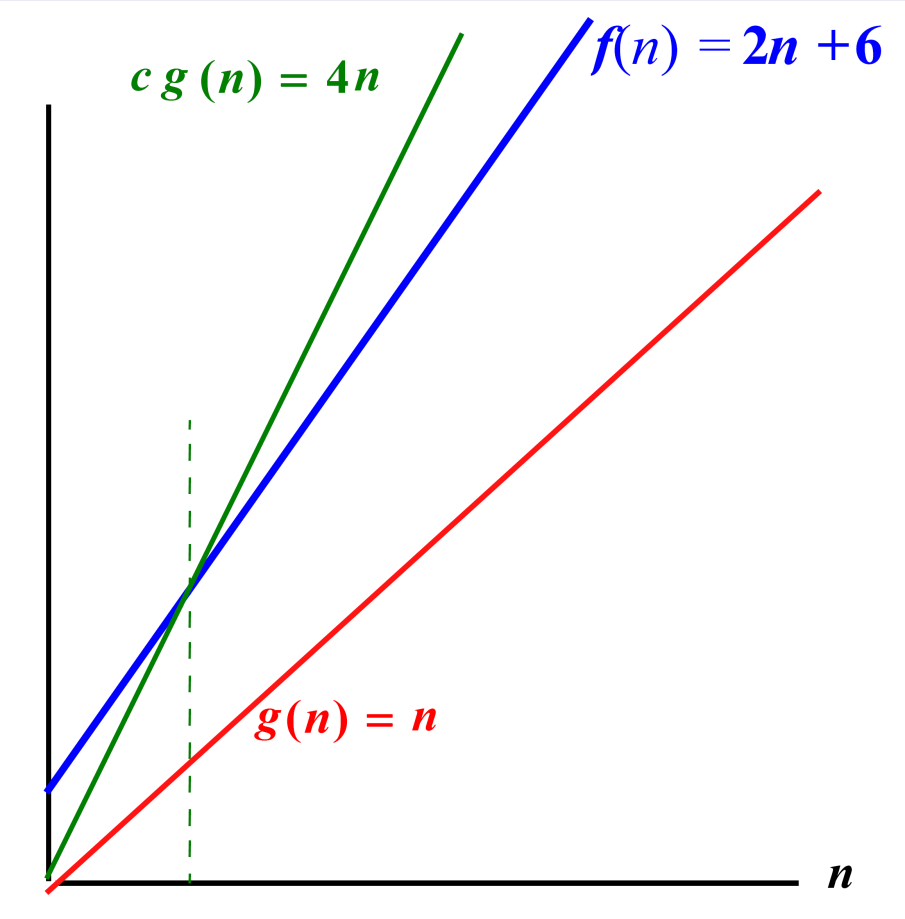
- We know:
  - Experimental approach – problems
  - Low level analysis – count operations
- Abstract even further
- Characterize an algorithm as a function of the "problem size"
- e.g.
  - Input data = array $\rightarrow$ problem size is N (length of array)
  - Input data = matrix $\rightarrow$ problem size is N $\times$ M

Asymptotic Notation

- Goal: to simplify analysis by getting rid of unneeded information (like "rounding" 1,000,001≈1,000,000)
- We want to say in a formal way $3n^2 \approx n^2$
- The "Big-Oh" Notation:
  - Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if and only if there are positive constants c and n_0 such that $f(n) \leq cg(n)$ for $n \geq n_0$
  - Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity. It is a member of a family of notations invented by Paul Bachmann, Edmund Landau, and others, collectively called Bachmann-Landau notation or asymptotic notation.

Graphic Illustration

- $f(n) = 2n + 6$
- Conf. def:
  - Need to find a function $g(n)$ and a const. c and a constant 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



More examples

- What about $f(n) = 4n^2$? Is it $O(n)$?
  - Find a c and n_0 such that $4n^2 < cn$ for any $n > n_0$
- $50n^3 + 20n + 4$ is $O(n^3)$
  - Would be correct to say is $O(n^3 + n)$?
    - Not useful, as n^3 exceeds by far n, for large values
  - Would be correct to say is $O(n^5)$?
    - OK, but $g(n)$ should be as close as possible to $f(n)$
- $3log(n) + log(log(n)) = O(?)$

Simple Rule: Drop lower order terms and constant factors

Big-Oh 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 $O(g_1(n) + g_2(n)) = max(O(g_1(n) + g_2(n))$
  - $f_1(n) \times f_2(n)$ is $O(g_1(n) \times g_2(n))$
- $log^k N$ is $O(N)$ for any constant k
- The relative growth rate of two functions can always be determined by computing their limit

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

- But using this method is almost always an overkill

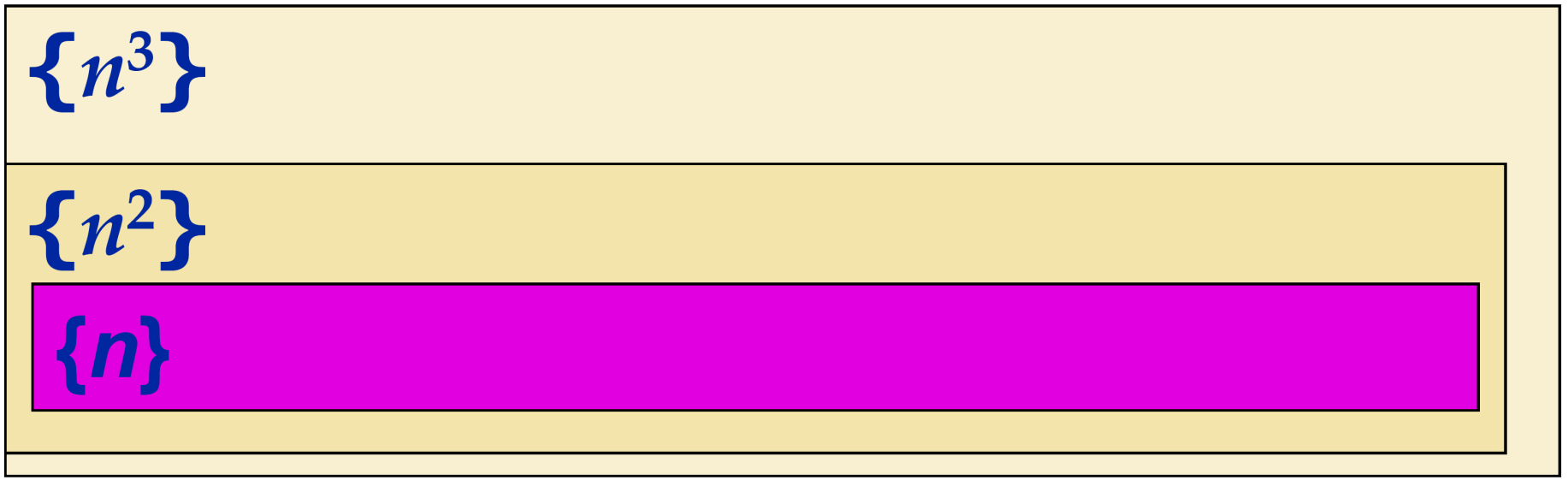
Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function.
- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$.
- 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

Classes of Functions

- Let $\{g(n)\}$ denote the class (set) of functions that are $O(g(n))$
- We have $\{n\} \subset \{n^2\} \subset \{n^3\} \subset \{n^4\} \subset \{n^5\} \subset \ldots$ where the containment is strict



Big-Oh Rules

- If is $f(n)$ 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 $O(n^2)$"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

Inappropriate Expressions

$f(n) \not\leq O(g(n))$
$f(n) \not\geq O(g(n))$

Properties of Big-Oh

- If $f(n)$ is $O(g(n))$ then $af(n)$ is $O(g(n))$ for any a.
- If $f(n)$ is $O(g(n))$ and $h(n)$ is $O(g'(n))$ then $f(n) + h(n)$ is $O(g(n) + g'(n))$
- If $f(n)$ is $O(g(n))$ and $h(n)$ is $O(g'(n))$ then $f(n)h(n)$ is $O(g(n)g'(n))$
- If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$ then $f(n)$ is $O(h(n))$

Properties of Big-Oh

- n^x is $O(a^n)$, for any fixed $x > 0$ and $a > 1$
  - An algorithm of order n to a certain power is better than an algorithm of order $a(> 1)$ to the power of n
- $log\ n^x$ is $O(log\ n)$, for $x > 0$ – how?
- $log^x\ n$ is $O(n^y)$ for $x > 0$ and $y > 0$
  - An algorithm of order $log\ n$ (to a certain power) is better than an algorithm of n raised to a power y.

Asymptotic analysis - terminology

- Special classes of algorithms:
  - logarithmic: $O(log\ n)$
  - linear: $O(n)$
  - quadratic: $O(^2)$
  - polynomial: $O(n^k), k \geq 1$
  - exponential: $O(a^n), n > 1$
- Polynomial vs. exponential ?
- Logarithmic vs. polynomial ?

Some Numbers

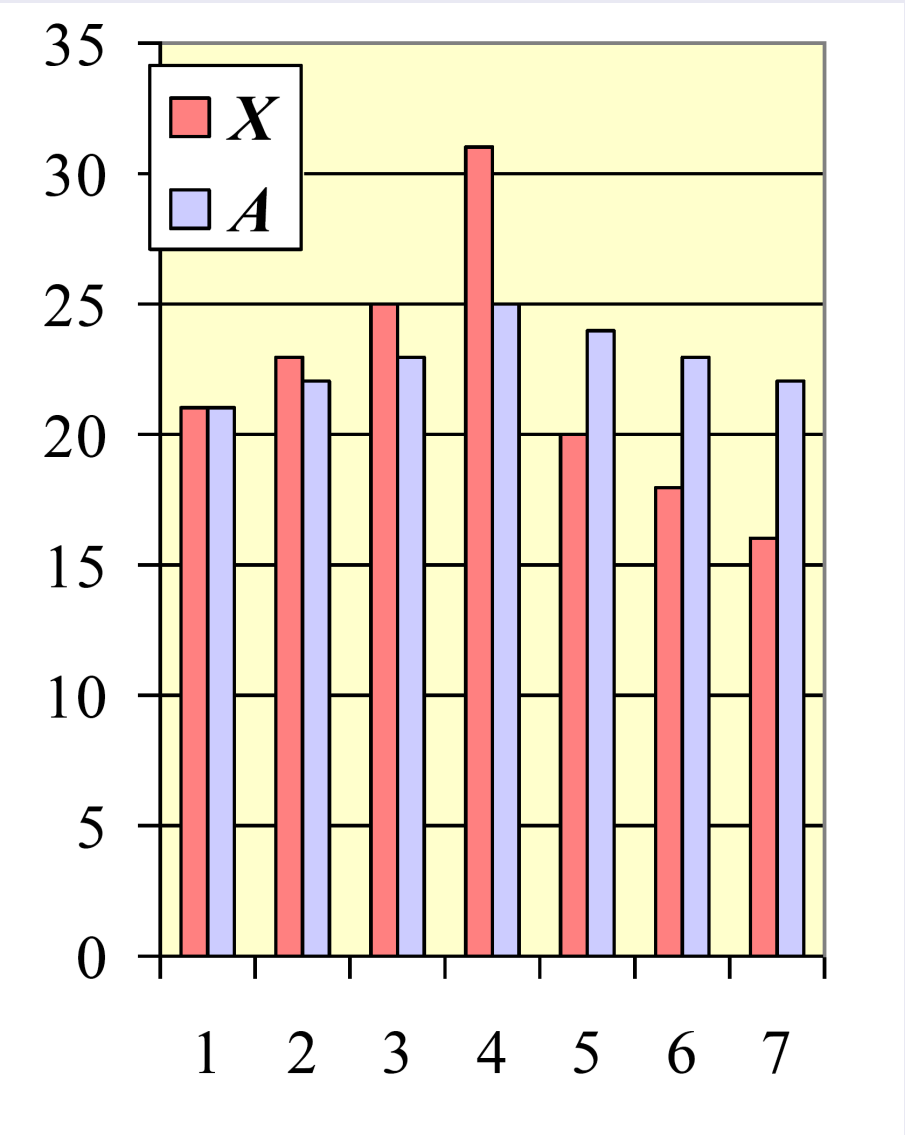
log n	n	n log n	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

Example:Computing Prefix Averages

- The i-th prefix average of an array `X` is average of the first (i + 1) elements of `X`

```
A[i] = (X[0] + X[1] +
...   + X[i])/(i+1)
```

- Computing the array `A` of prefix averages of another array `X` has applications to financial analysis



Prefix Averages (Quadratic)

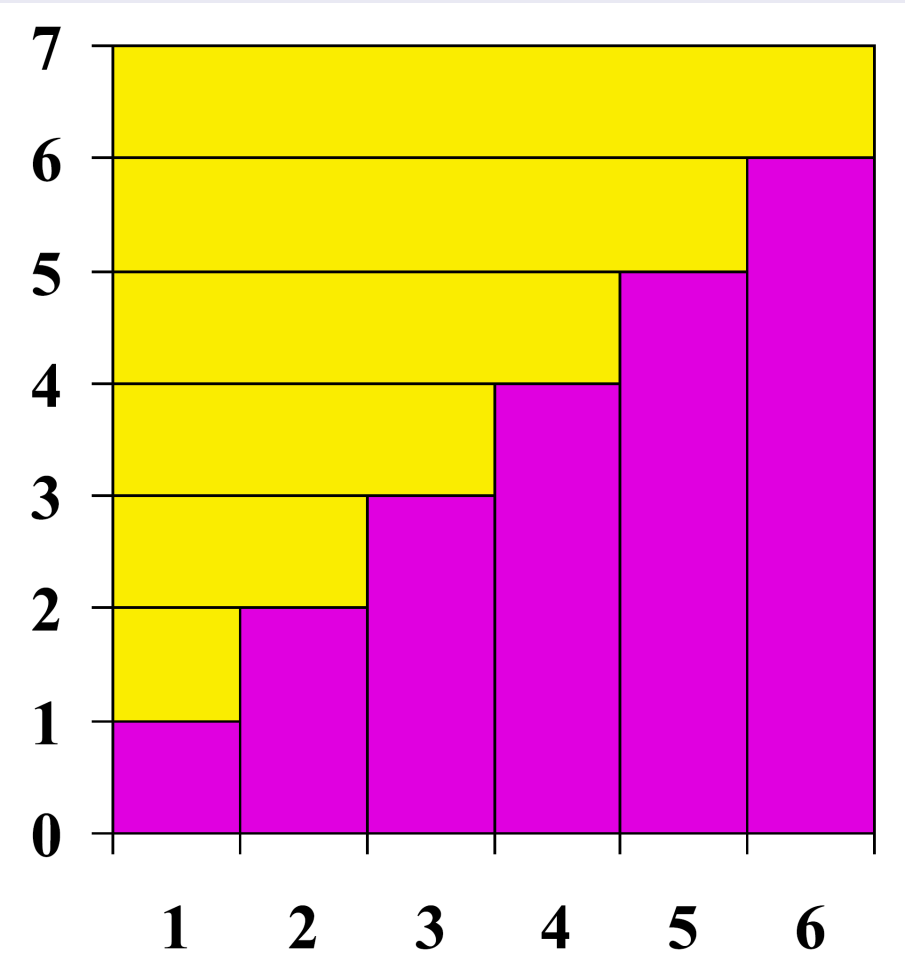
- The following algorithm computes prefix averages in quadratic time by applying the definition.

Algorithm 3 prefixAverages1(X, n)

		#operations
1:	**Input** *array X of n integers*.	
2:	**Output**	
	array A of prefix averages.	
3:	$A \leftarrow$ *new array of n integers*.	n
4:	**for** $i \leftarrow 0$ *to* $n-1$ **do**	n
5:	$s \leftarrow X[0]$	n
6:	**for** $j \leftarrow 1$ *to* i **do**	1+2+...+(n-1)
7:	$s \leftarrow s + X[j]$	1+2+...+(n-1)
8:	$A[i] \leftarrow s/(i+1)$	n
	return A	1

Arithmetic Progression

- The running time of *prefixAverages1* is $O(1 + 2 + \cdots + n)$
- The sum of the first n integers is $n(n+1)/2$
  - There is a simple visual proof of this fact
- Thus, algorithm *prefixAverages1* runs in $O(n^2)$ time



Prefix Averages (Linear)

- The following algorithm computes prefix averages in linear time by keeping a running sum.

Algorithm 4 prefixAverages2(X, n)

		#operations
1:	**Input** *array X of n integers*.	
2:	**Output** *array A of prefix averages*.	
3:	$A \leftarrow$ *new array of n integers*.	n
4:	$s \leftarrow 0$	1
5:	**for** $i \leftarrow 0$ *to* $n-1$ **do**	n
6:	$s \leftarrow s + X[i]$	n
7:	$A[i] \leftarrow s/(i+1)$	n
	return A	n
		1

- Algorithm *prefixAverages2* runs in $O(n)$ time

How many operations?

Table: of functions with respect to input n, assume that each primitive operation takes one microsecond (1 second = 10^6 microsecond).

$O(g(n))$	1 Second	1 Hour	1 Month	1 Century
$log_2 n$	$\approx 10^{300000}$	$\approx 10^{10^9}$	$\approx 10^{0.8\times 10^{12}}$	$\approx 10^{10^{15}}$
$\sqrt{n}$	$\approx 10^{12}$	$\approx 1.3 \times 10^{19}$	$\approx 6.8 \times 10^{24}$	$\approx 9.7 \times 10^{30}$
n	10^6	3.6×10^9	$\approx 2.6 \times 10^{12}$	$\approx 3.12 \times 10^{15}$
$nlog_2 n$	$\approx 10^5$	$\approx 10^9$	$\approx 10^{11}$	$\approx 10^{14}$
n^2	1000	6×10^4	$\approx 1.6 \times 10^6$	$\approx 5.6 \times 10^7$
n^3	100	≈ 1500	≈ 14000	≈ 1500000
2^n	19	31	41	51
$n!$	9	12	15	17

Running Time Calculations

General Rules

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

Some Examples

```
Case1: for (i=0; i<n; i++)
           for (j=0; j<n; j++)
               k++;
```
$O(n^2)$

```
Case 2: for (i=0; i<n; i++)
            k++;
        for (i=0; i<n; i++)
            for (j=0; j<n; j++)
                k++;
```
$O(n^2)$

```
Case 3: for (int i=0; i<n; i++)
            for (int j=0; j<n; j++)
                k+=1;
```
$O(n^2)$

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(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;
}
```

$O(n^3)$

The Second Algorithm

```
int MaxSubSum2(const 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(const 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;
```

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

$O(n)$

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;
}
```

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

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

Analyzing recursive algorithms

```
function foo(param A, param B) {
        statement_1;
        statement_2;
        if (termination condition)
                return;
        foo(A_1, B_1);
}
```

Solving recursive equations by repeated substitution

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

$= T(n/4) + c + c$ substitute for $T(n/4)$

$= T(n/8) + c + c + c$

$= T(n/2^3) + 3c$ in more compact form

$= \ldots$

$= T(n/2^k) + kc$ "inductive leap"

$T(n) = T(n/2^{log\ n}) + c\log n$ "choose $k = \log n$"

$= T(n/n) + c\log n$

$= T(1) + c\log n = b + c\log n = \Theta(log\ n)$

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
$T(n) = \Theta(\log n)$

The Fourth Algorithm

```
int MaxSubSum4(const 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;
}
```

$O(n)$

Problem:

- Order the following functions by their asymptotic growth rates
  - $nlogn$
  - $logn^3$
  - n^2
  - $n^{2/5}$
  - 2^{logn}
  - $log(logn)$
  - $Sqr(logn)$

Back to the original question

- Which solution would you choose?
  - $O(n^2)$ vs. $O(n)$
- Some math . . .
  - properties of logarithms:
    - $log_b(xy) = log_b x + log_b y$
    - $log_b(x/y) = log_b x - log_b y$
    - $log_b x^a = a log_b x$
    - $log_b a = log_x a / log_x b$
  - properties of exponentials:
    - $a^{(b+c)} = a^b a^c$
    - $a^{bc} = (a^b)^c$
    - $a^b / a^c = a^{(b-c)}$
    - $b = a^{log_a b}$
    - $b^c = a^{c \times log_a b}$

“Relatives” of Big-Oh

- “Relatives” of the Big-Oh
  - $\Omega(f(n))$: Big Omega – asymptotic lower bound
  - $\Theta(f(n))$: Big Theta – asymptotic tight bound
- Big-Omega – think of it as the inverse of $O(n)$
  - $g(n)$ is $\Omega(f(n))$ if $f(n)$ is $O(g(n))$
- Big-Theta – combine both Big-Oh and Big-Omega
  - $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $g(n)$ is $\Omega(f(n))$
- Consider the difference:
  - $3n + 3$ is $O(n)$ and is $\Theta(n)$
  - $3n + 3$ is $O(n^2)$ but is not $\Theta(n^2)$

More “relatives”

- 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 $o(n^2)$
- $2n + 3$ is $o(n)$?

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 $g(n)$

Important Series

- Sum of squares:

$$S(N) = 1 + 2 + \cdots + N = \sum_{i=1}^{N} i = N\frac{(1+N)}{2}$$

- Sum of exponents:

$$\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} \approx \frac{N^3}{3} \text{ for large } N$$

- Geometric series:
  - Special case when $A = 2$
    - $2^0 + 2^1 + 2^2 + \cdots + 2^N = 2^{N+1} - 1$

$$\sum_{i=1}^{N} i^k \approx \frac{N^{k+1}}{|k+1|} \text{ for large } N \text{ and } k \neq -1$$

Problem

- Running time for finding a number in a sorted array [binary search]
  - Pseudo-code
  - Running time analysis