NANYANG TECHNOLOGICAL UNIVERSITY

CE2001/ CZ2001: Algorithms

Analysis of Algorithms
(Recursive Functions)

Dr. Loke Yuan Ren

NANYANG TECHNOLOGICAL UNIVERSITY

Learning Objectives

At the end of this lecture, students should be able to:

- Express the time complexity of recursive functions in terms of recurrent equations
- Solve the recurrent equations to obtain the time complexity function

NANYANG TECHNOLOGICAL UNIVERSITY

Time Complexity of Recursive Functions

For a simple recursive function, determine:

1. Number of primitive operations for each recursive call
2. Number of recursive calls

NANYANG TECHNOLOGICAL UNIVERSITY

Time Complexity of Recursive Functions

For a simple recursive function, determine:

1. Number of primitive operations for each recursive call
2. Number of recursive calls

Example 1:

```
int factorial (int n)
{       if (n == 1) return 1;
        else return n * factorial(n-1);
}
```

Time Complexity of Recursive Functions

For a simple recursive function, determine:

1. Number of primitive operations for each recursive call
2. Number of recursive calls

Example 1:

```
int factorial (int n)
{       if (n == 1) return 1;
        else return n * factorial(n-1);
}
```


Time Complexity of Recursive Functions

For a simple recursive function, determine:

1. Number of primitive operations for each recursive call
2. Number of recursive calls

Example 1:

```
int factorial (int n)
{       if (n == 1) return 1;
        else return n * factorial(n-1);
}
```

n is reduced by 1 at each recursive call

Time Complexity of Recursive Functions

For a simple recursive function, determine

1. Number of primitive operations for each recursive call
2. Number of recursive calls

Example 1:

```
int factorial (int n)
{       if (n == 1) return 1;
        else return n * factorial(n-1);
}
```

n is reduced by 1 at each recursive call

For each recursive call when n > 1: c_1
For the last recursive call when n is 1: c_2
Number of recursive calls: n-1 (non-terminating case) + 1
Time complexity: $c_1*(n-1) + c_2$ (this is roughly $c_1 n$)

Time Complexity of Recursive Functions

Example 2:

```
int count (int array[ ], int n, int a)
{ // count how many times item 'a' appears in 'array'
    if  (n == 1)
            if (array[0] == a)
                return 1;
            else return 0;
    if (array[0] == a)
            return 1 + count (&array[1], n-1, a);
    else
            return count (&array[1], n-1, a);
}
```

Time Complexity of Recursive Functions

Example 2:

```
int count (int array[ ], int n, int a)
{ // count how many times item 'a' appears in 'array'
    if (n == 1)
            if (array[0] == a)
                return 1;
            else return 0;
    if (array[0] == a)
            return 1 + count (&array[1], n-1, a);
    else
            return count (&array[1], n-1, a);
}
```


Time Complexity of Recursive Functions

Example 2:

```
int count (int array[ ], int n, int a)
{ // count how many times item 'a' appears in 'array'
    if (n == 1)
            if (array[0] == a)
                return 1;
            else return 0;
    if (array[0] == a)
            return 1 + count (&array[1], n-1, a);
    else
            return count (&array[1], n-1, a);
}
```

Address of next array element

Time Complexity of Recursive Functions

Example 2:

```
int count (int array[ ], int n, int a)
{ // count how many times item 'a' appears in 'array'
    if (n == 1)
            if (array[0] == a)
                return 1;
            else return 0;
    if (array[0] == a)
            return 1 + count (&array[1], n-1, a);
    else
            return count (&array[1], n-1, a);
}
```


Time Complexity of Recursive Functions

Consider the number of comparisons between an array element and a:

$W_1 = 1$

$W_n = 1 + W_{n-1}$

$W_n = 1 + W_{n-1}$

$= 1 + 1 + W_{n-2}$

$= 1 + 1 + 1 + W_{n-3}$

....

$= 1 + 1 + \ldots + 1 + W_{n-(n-1)}$

$= n$

W_n is the number of comparisons in count (array[], n, a)

W_{n-1} is the number of comparisons in count (&array[1], n-1, a)

Alternative notation:
W(1) = 1
W(n) = 1 + W(n-1)

n-1 1's

Method of Backward Substitutions

Time Complexity of Recursive Functions

Example 3: Consider the number of multiplications

```
preorder (simple_t * tree)
{
    if (tree != NULL)  {
            tree->item *= 10;
            preorder (tree->left);
            preorder (tree->right);
                }
}
```


Time Complexity of Recursive Functions

Example 3: Consider the number of multiplications

```
preorder (simple_t * tree)
{
    if (tree != NULL)  {
            tree->item *= 10;
            preorder (tree->left);
            preorder (tree->right);
                }
}
```


Time Complexity of Recursive Functions

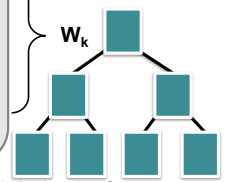
Example 3: Consider the number of multiplications

```
preorder (simple_t * tree)
{
    if (tree != NULL)  {
            tree->item *= 10;
            preorder (tree->left);
            preorder (tree->right);
                }
}
```


Time Complexity of Recursive Functions

Example 3: Consider the number of multiplications

```
preorder (simple_t * tree)
{
    if (tree != NULL)  {
            tree->item *= 10;      ⟶ 1
            preorder (tree->left);     ⟶ W_L
            preorder (tree->right);    ⟶ W_R
                }
}
```



Let W_k be the no. of multiplications for a tree with **k** depth of the tree.

$W_0 = 0$

$W_k = 1 + W_L + W_R$

Alternative notation:
W(0) = 0
W(n) = 1 + W(L) + W(R)

Time Complexity of Recursive Functions

- Assume that it is a complete binary tree
- The number of nodes is $2^k - 1$ (k is the depth of the tree)

$W_0 = 0$

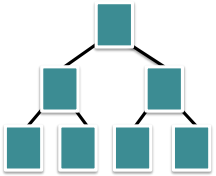
$W_1 = 1$

$W_2 = 1 + W_1 + W_1 = 1 + 2 = 3$

$W_3 = 1 + W_2 + W_2 = 1 + 2\ (1+2) = 1+2+4 = 7$

…

$W_k = 1 + W_{k-1} + W_{k-1} = 1 + 2 + 4 + \ldots + 2^{k-1} = 2^k - 1$



- The # of recursive calls is a geometric series
- Prove the hypothesis can be done by mathematical induction
- Method of Forward Substitutions

Time Complexity of Recursive Functions

```
int fibonacci (int n){
    if( n < 1) return 0;
    if(n == 1 || n ==2) return 1;
    return fibonacci (n-1) + fibonacci (n-2);
}
```

Example 4: Consider the number of function calls

W(n) = W(n-1) + W(n-2)+1

W(1) = W(2) = 1

Neither Forward Substitutions nor Backward Substitutions are able to solve the time complexity.

Approximation Solution

```
int fibonacci (int n){
    if( n < 1) return 0;
    if(n == 1 || n ==2) return 1;
    return fibonacci (n-1) + fibonacci (n-2);
}
```

W(n) = W(n-1) + W(n-2)+1
W(1) = W(2) = 1

Example 4: Consider the number of function calls

$$W(n) = W(n-1) + W(n-2) + 1$$

$$\approx 2W(n-1) + 1$$

$$= 2\ (2\ \text{W(n-2)}+1) + 1$$

$$= 2\ (2\ (\ 2\text{W(n-3)}+1) + 2 + 1$$

…

$$= 2^{n-3}\left(2W\left(n-(n-2)\right)+1\right) + 2^{n-4} + 2^{n-5} + \ldots + 4 + 2 + 1$$

$$= 2^{n-2} + 2^{n-3} + 2^{n-4} + 2^{n-5} + \ldots + 4 + 2 + 1$$

$$= 2^{n-1} - 1$$

$$S_n = a + ar + ar^2 + \ldots + ar^{n-1}$$

$$S_n = \frac{a(1-r^n)}{1-r}$$

Time Complexity of Recursive Functions

```
int fibonacci (int n){
    if( n < 1) return 0;
    if(n == 1 || n ==2) return 1;
    return fibonacci (n-1) + fibonacci (n-2);
}
```

Example 4: Consider the number of function calls

W(n) = W(n-1) + W(n-2)+1

W(1) = W(2) = 1

Neither Forward Substitutions nor Backward Substitutions are able to solve the time complexity.

Can we solve it exactly?

Second-Order Linear Recurrences with Constant Coefficients

$$aW(n) + b\,W(n-1) + c\,W(n-2) = f(n)$$

- Constant coefficients => a, b and c are constant
- Second order => W(n) and W(n-2) are two position apart in the sequence
- f(n)=
  - 0 => homogeneous
  - Otherwise => non-homogeneous

Second-Order Linear Recurrences with Constant Coefficients

$$aW(n) + b\,W(n-1) + c\,W(n-2) = f(n)$$

- Its characteristic equation is

$$ar^2 + br + c = 0$$

- The roots of the characteristic equation are

$$r = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

1. Two real and distinct root: r_+ and r_- $(b^2 - 4ac > 0)$
2. A real double root: $r_+ == r_-$ $(b^2 - 4ac = 0)$
3. Two distinct root: $b^2 - 4ac < 0$

Second-Order Linear Recurrences with Constant Coefficients

$$aW(n) + b\,W(n-1) + c\,W(n-2) = 0$$

- Its characteristic equation is

$$ar^2 + br + c = 0$$

- The roots of the characteristic equation are

$$r = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

1. Two real and distinct root: r_+ and r_- $(b^2 - 4ac > 0)$
2. A real double root: $r_+ == r_-$ $(b^2 - 4ac = 0)$
3. Two distinct root: $b^2 - 4ac < 0$

A linear combination of two geometric sequence

Homogeneous solution

$W^h(n) = \alpha r_+^n + \beta r_-^n$

$W^h(n) = \alpha r^n + \beta n r^n$

$W^h(n) = \gamma^n[\alpha \cos n\theta + i\beta \sin n\theta]$

Remarks: Case 1 and case 3 are just real and complex roots. The solution is just a trigonometric form of complex number

Second-Order Linear Recurrences with Constant Coefficients

$$aW(n) + b\,W(n-1) + c\,W(n-2) = kr_p^n$$

- The general solution

$$W(n) = W^h(n) + W^p(n)$$

Homogeneous solution

Particular solution

- The particular solution, $W^p(n)$

1. Not equal to any root: $r_p \neq r_+, r_p \neq r_-$
2. Equal to one of the roots: $r_p = r_+$ or $r_p = r_-$
3. Equal to a double root: $r_p = r_+ = r_-$

Particular solution

$W^p(n) = Ar_p^n$

$W^p(n) = Anr_p^n$

$W^p(n) = An^2 r_p^n$

Time Complexity of Recursive Functions

```
int fibonacci (int n){
    if( n < 1) return 0;
    if(n == 1 || n ==2) return 1;
    return fibonacci (n-1) + fibonacci (n-2);
}
```

Example 4: Consider the number of function calls

W(n) = W(n-1) + W(n-2)+1

W(1) = W(2) = 1

Neither Forward Substitutions nor Backward Substitutions are able to solve the time complexity.

Can we solve it exactly?

W(n) - W(n-1) - W(n-2) = 1

Time Complexity of Recursive Functions

W(n) - W(n-1) - W(n-2) = 1

$$W(n) = W^h(n) + W^p(n)$$

$W^h(n)$:

The root of the characteristic equation:

$$r = \frac{-1 \pm \sqrt{5}}{2}$$

The homogeneous solution is

$$W^h(n) = \alpha(\frac{1+\sqrt{5}}{2})^n + \beta(\frac{1-\sqrt{5}}{2})^n$$

$W^p(n)$:

$f(n) = 1 \cdot 1^n$

Not equal to any root

The particular solution is

$W^p(n) = A$

$$W(n) = W^h(n) + W^p(n)$$

$$W(n) = \alpha(\frac{1+\sqrt{5}}{2})^n + \beta(\frac{1-\sqrt{5}}{2})^n + A$$

$$W^p(n) - W^p(n-1) - W^p(n-2) = 1$$

$$A - A - A = 1$$

$$A = -1$$

$$W(n) = \alpha(\frac{1+\sqrt{5}}{2})^n + \beta(\frac{1-\sqrt{5}}{2})^n - 1$$

Substitute W(1) = W(2) = 1,

$$W(n) = \frac{2}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n - \frac{2}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^n - 1$$

Time Complexity of Recursive Functions

```
int fibonacci (int n){
    if( n < 1) return 0;
    if(n == 1 || n ==2) return 1;
    return fibonacci (n-1) + fibonacci (n-2);
}
```

n	W(n)
1	1
2	1
3	3
4	5
5	9
6	15
7	25
8	41
9	67
10	109
11	177
12	287

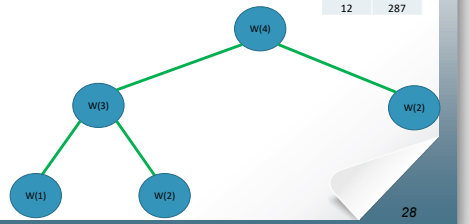
How many function calls are there in this algorithm?

W(n) = W(n-1) + W(n-2) + 1

W(1) = W(2) = 1

$$W(n) = \frac{2}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n - \frac{2}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^n - 1$$

$$\approx 1.618^n$$



Summary

Analysing Recursive Functions by:

- Expressing the time complexity of recursive functions in terms of recurrent equations
- Solving the recurrent equations to obtain the time complexity function
  - Method of forward substitutions
  - Method of backward substitutions
  - 2nd order linear recurrences

NANYANG TECHNOLOGICAL UNIVERSITY

CE2001/ CZ2001: Algorithms

Analysis of Algorithms

(Order of Growth of Time Complexity Functions)

Dr. Loke Yuan Ren

NANYANG TECHNOLOGICAL UNIVERSITY

Learning Objectives

At the end of this lecture, students should be able to:

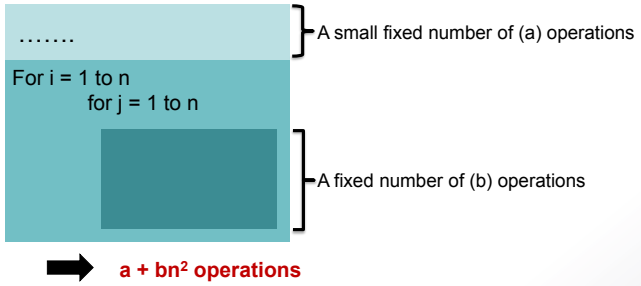
- Appreciate the order of growth of different time complexity functions
- Analyse the impact of faster algorithms versus faster computers

NANYANG TECHNOLOGICAL UNIVERSITY

Growth Rate of Time Complexity

We are concerned with the growth rate of resources

1. Constants are not significant when n is large
2. Multipliers are not important for relative growth rate



➡ **$a + bn^2$ operations**

NANYANG TECHNOLOGICAL UNIVERSITY

How Functions Grow?

Algo	1	2	3	4	5	6
Operation (μsec)	$13n$	$13n\log_2 n$	$13n^2$	$130n^2$	$13n^2+10^2$	2^n

How Functions Grow?

Algo	1	2	3	4	5	6
Operation (µsec)	13n	$13n\log_2 n$	$13n^2$	$130n^2$	$13n^2+10^2$	2^n

Input (n) — Time Cost (Second)

10	.00013	.00043	.0013	.013	.0014	.001024
100	.0013	.0086	.13	1.3	.1301	$4*10^{16}$ yrs
10^4	.13	1.73	22 mins	3.61 hrs	22 mins	
10^6	13	259	150 days	1505 days	150 days	

How Functions Grow?

Algo	1	2	3	4	5	6
Operation (µsec)	13n	$13n\log_2 n$	$13n^2$	$130n^2$	$13n^2+10^2$	2^n

Input (n) — Time Cost (Second)

10	.00013	.00043	.0013	.013	.0014	.001024
100	.0013	.0086	.13	1.3	.1301	$4*10^{16}$ yrs
10^4	.13	1.73	22 mins	3.61 hrs	22 mins	
10^6	13	259	150 days	1505 days	150 days	

How Functions Grow?

Algo	1	2	3	4	5	6
Operation (µsec)	13n	$13n\log_2 n$	$13n^2$	$130n^2$	$13n^2+10^2$	2^n

Input (n) — Time Cost (Second)

10	.00013	.00043	.0013	.013	.0014	.001024
100	.0013	.0086	.13	1.3	.1301	$4*10^{16}$ yrs
10^4	.13	1.73	22 mins	3.61 hrs	22 mins	
10^6	13	259	150 days	1505 days	150 days	

How Functions Grow?

Algo	1	2	3	4	5	6
Operation (µsec)	13n	13nlogn	$13n^2$	$130n^2$	$13n^2+10^2$	2^n

Input (n) — Time Cost (Second)

10	.00013	.00043	.0013	.013	.0014	.001024
100	.0013	.0086	.13	1.3	.1301	$4*10^{16}$ yrs
10^4	.13	1.73	22 mins	3.61 hrs	22 mins	
10^6	13	259	150 days	1505 days	150 days	

How Functions Grow?

Algo	1	2	3	4	5	6
Operation (µsec)	13n	$13n\log_2 n$	$13n^2$	$130n^2$	$13n^2+10^2$	2^n

Input (n)	Time Cost (Second)					
10	.00013	.00043	.0013	.013	.0014	.001024
100	.0013	.0086	.13	1.3	.1301	$4*10^{16}$ yrs
10^4	.13	1.73	22 mins	3.61 hrs	22 mins	
10^6	13	259	150 days	1505 days	150 days	

How Functions Grow?

Algo	1	2	3	4	5	6
Operation (µsec)	13n	$13n\log_2 n$	$13n^2$	$130n^2$	$13n^2+10^2$	2^n

Input (n)	Time Cost (Second)					
10	.00013	.00043	.0013	.013	.0014	.001024
100	.0013	.0086	.13	1.3	.1301	$4*10^{16}$ yrs
10^4	.13	1.73	22 mins	3.61 hrs	22 mins	
10^6	13	259	150 days	1505 days	150 days	

How Functions Grow?

Algo	1	2	3	4	5	6
Operation (µsec)	13n	$13n\log_2 n$	$13n^2$	$130n^2$	$13n^2+10^2$	2^n

Input (n)	Time Cost (Second)					
10	.00013	.00043	.0013	.013	.0014	.001024
100	.0013	.0086	.13	1.3	.1301	$4*10^{16}$ yrs
10^4	.13	1.73	22 mins	3.61 hrs	22 mins	
10^6	13	259	150 days	1505 days	150 days	

How Functions Grow?

Algo	1	2	3	4	5	6
Operation (µsec)	13n	$13n\log_2 n$	$13n^2$	$130n^2$	$13n^2+10^2$	2^n

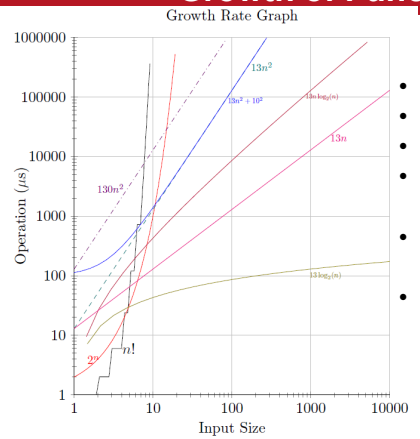
Input (n)	Time Cost (Second)					
10	.00013	.00043	.0013	.013	.0014	.001024
100	.0013	.0086	.13	1.3	.1301	$4*10^{16}$ yrs
10^4	.13	1.73	22 mins	3.61 hrs	22 mins	
10^6	13	259	150 days	1505 days	150 days	

How Functions Grow?

Algo	1	2	3	4	5	6
Operation (μsec)	13n	$13n\log_2 n$	$13n^2$	$130n^2$	$13n^2+10^2$	2^n

Input (n)	Time Cost (Second)					
10	.00013	.00043	.0013	.013	.0014	.001024
100	.0013	.0086	.13	1.3	.1301	$4*10^{16}$ yrs
10^4	.13	1.73	22 mins	3.61 hrs	22 mins	
10^6	13	259	150 days	1505 days	150 days	

Growth of Functions



- n! is the fastest growth
- 2^n is the second
- 13n is linear
- $13\log_2 n$ is the slowest

- 10^2 can be ignored when n is large
- $13n^2$ and $130n^2$ have similar growth.
  - $130n^2$ slightly faster

Faster Computer Or Faster Algorithm?

- You have devised an algorithm to solve a problem
- Algorithm takes too long to run
- Get a faster computer or improve the algorithm?
- Problem size does not remain the same with faster computer
- People want to do more work with faster computer
- Hence, problem remains

How Would Faster Computer Help?

Scenario

- Suppose old computer executes 10,000 basic operations per hour
- New computer, which is 10 times faster, executes 100,000 basic operations per hour
- If old computer solves a problem of size n in one hour, what is the largest problem the new computer can solve in one hour?

How Would Faster Computer Help?

Findings

- For linear complexity problem, the improvement is 10 times
- For all faster-growing than linear (harder) problems , the improvement is poorer than the linear problems'
- For exponential complexity problem, there is hardly any improvement

How Would Faster Computer Help?

f(n)	n	n'	Change	n'/n
10n	1,000	10,000	n'=10n	10
20n	500	5,000	n'=10n	10
$5n \log_2 n$	250	1,842	3.16n<n'<10n	7.37
$2n^2$	70	223	n'=3.16n	3.16
2^n	13	16	n'=n+3	1.23

How Would Faster Computer Help?

f(n)	n	n'	Change	n'/n
				10
		5,0		10
				7.37
$2n^2$	70	223	n'=3.16n	3.16
2^n	13	16	n'=n+3	1.23

n: Largest problem size solved by old computer, which performs 10,000 operations in 1 hour

n': Largest problem size solved by new computer, which performs 100,000 operations in 1 hour

How Would Faster Computer Help?

f(n)	n	n'	Change	n'/n
10n	1,000	10,000	n'=10n	10
20n	500	5,000	n'=10n	10
$5n \log_2 n$	250	1,842	3.16n<n'<10n	7.37
$2n^2$	70	223	n'=3.16n	3.16
2^n	13	16	n'=n+3	1.23

How Would Faster Computer Help?

f(n)	n	n'	Change	n'/n
10n	**1,000**	**10,000**	**n'=10n**	**10**
20n	500	5,000	n'=10n	10
5n $\log_2$ n	250	1,842	3.16n<n'<10n	7.37
$2n^2$	70	223	n'=3.16n	3.16
2^n	13	16	n'=n+3	1.23

How Would Faster Computer Help?

f(n)	n	n'	Change	n'/n
10n	1,000	10,000	n'=10n	10
20n	**500**	**5,000**	**n'=10n**	**10**
5n $\log_2$ n	250	1,842	3.16n<n'<10n	7.37
$2n^2$	70	223	n'=3.16n	3.16
2^n	13	16	n'=n+3	1.23

How Would Faster Computer Help?

f(n)	n	n'	Change	n'/n
10n	1,000	10,000	n'=10n	10
20n	500	5,000	n'=10n	10
5n $\log_2$ n	**250**	**1,842**	**3.16n<n'<10n**	**7.37**
$2n^2$	70	223	n'=3.16n	3.16
2^n	13	16	n'=n+3	1.23

How Would Faster Computer Help?

f(n)	n	n'	Change	n'/n
10n	1,000	10,000	n'=10n	10
20n	500	5,000	n'=10n	10
5n $\log_2$ n	250	1,842	3.16n<n'<10n	7.37
$2n^2$	**70**	**223**	**n'=3.16n**	**3.16**
2^n	13	16	n'=n+3	1.23

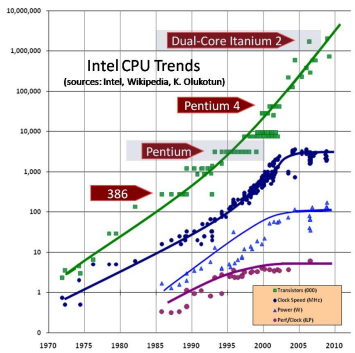
How Would Faster Computer Help?

f(n)	n	n'	Change	n'/n
10n	1,000	10,000	n'=10n	10
20n	500	5,000	n'=10n	10
5n $\log_2$ n	250	1,842	3.16n<n'<10n	7.37
$2n^2$	70	223	n'=3.16n	3.16
2^n	**13**	**16**	**n'=n+3**	**--**

How About Improving The Algorithm?

- **Suppose complexity function is n^2**
  - When n = 1024, it takes 1024×1024 = 1,048,576 primitive steps to run
- **Suppose complexity function is reduced to n $\log_2$ n**
  - Then for n = 1024, it takes 10,240 primitive steps to run
  - More than a factor of 100 times improvement
  - What is the improvement when n = 2048?
  - n^2 = 2048 x 2048 = 4,194,304
  - $n\log_2 n$ = 2048 x $\log_2$ 2048 = 22,528
- **A factor of 200 times improvement!**

Moore's Law



- number of transistors doubles every 2 years
- The cost of computers is halved
- Clock speed is limited at 5Ghz
- It is always good that you have a more efficient algorithm.

Summary

- Order of Growth of Functions
- When a faster computer can help
- Importance of having faster algorithms