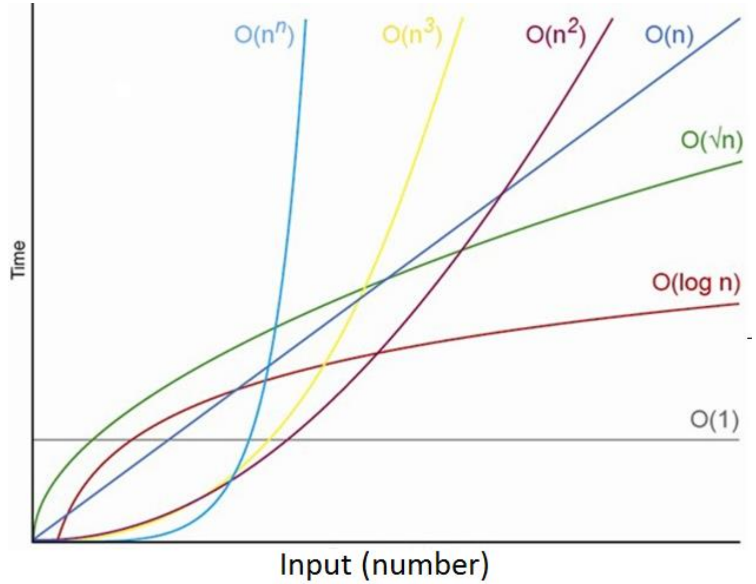
Quick REVIEW of Last Lecture

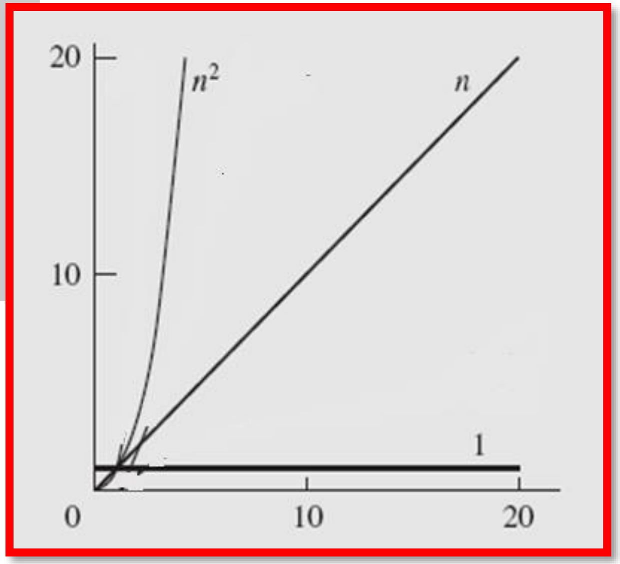
Time Complexity



Analysis of Loop

Algorithm	Cost (time complexity)
`i=0;`	**1**
`sum = 0;`	**1**
`while(i<N ){`	**N+1**
`sum ++;`	**N**
`i++`	**N**
`}`	

```
T(N) = 1 + 1 + N+1 + N + N
T(N) = 3N+3
```



We wish to FIND

- What is the effect of **constant** and **low order terms** in *T(N)* **?**

NESTED Loop

Code	Cost	Times
`i=1;`	1	1
`sum = 0;`	1	1
`while (i <= n) {`	1	n+1
`    j=1;`	1	n
`    while (j <= n) {`	1	n*(n+1)
`        sum += i;`	1	n*n
`        j++;`	1	n*n
`    }`		
`    i++;`	1	n
`}`		

T(N) = $1 + 1 + (n+1) + n + n*(n+1)+n*n+ n*n+ n = \mathbf{3n^2 + 3n + 3}$

➔ The time required for this algorithm is **proportional to n^2**

We wish to FIND

- What is the effect of **constant** and **low order terms** in *T(N)* **?**

Example: Effect of low order terms

- If **T(n) = 7n+100**
- What is T(n) for different values of n???

n	T(n)	Comment
1	107	Contributing factor is 100
5	135	Contributing factor is 7n and 100
10	170	Contributing factor is 7n and 100
100	800	Contribution of 100 is small
1000	7100	Contributing factor is 7n
10000	70100	Contributing factor is 7n
10^6	7000100	What is the contributing factor????

***DEDUCTION**: When approximating T(n) we can IGNORE the 100 term for very large value of n and say that T(n) can be approximated by 7(n)*

Example 2

$$T(n) = n^2 + 100n + \log_{10}n + 1000$$

n	T(n)	n^2		100n		$\log_{10}n$		1000	
		Val	%	Val	%	Val	%	Val	%
1	1101	1	0.1%	100	9.1%	0	0%	1000	90.8%
10	2101	100	5.8%	1000	47.6%	1	0.05%	1000	47.6%
100	21002	10000	47.6%	10000	47.6%	2	0.99%	1000	4.76%
10^5	10,010,001,005	10^{10}	99.9%	10^7	.099%	5	0.0%	1000	0.00%

When approximating T(n) we can ***IGNORE the last 3 terms*** *and say that T(n) can be approximated by* n^2

Rate of Growth

- Consider the example of buying *Gold and Metal jewelry*

 + 

Cost: cost_of_gold + cost_of_metal

Cost ~ cost_of_gold (approximation)

Partial SUM

	Cost
`int sum( int n )`	0
`{`	
`int partialSum;`	
`1 partialSum = 0;`	1
`2` **`for( int i = 1; i <= n; ++i )`**	i=1 cost is 1, i<=N cost is N+1 ++i cost is N
`3` **`partialSum += i * i * i;`**	4 ops 1 assignment, 3 multiplication, 1 add Total 4 cost for line 3
`4 return partialSum;`	0
`}`	

T(N)= 6N+4

If we had to perform all this work everytime we needed to analyze a program, the task would quickly **become infeasible**.

Problems with T(n)

T(n) is difficult to calculate

T(n) is also not very meaningful as step size is not exactly defined

T(n) is usually very complicated so we need an **approximation of T(n)….close to T(n).**

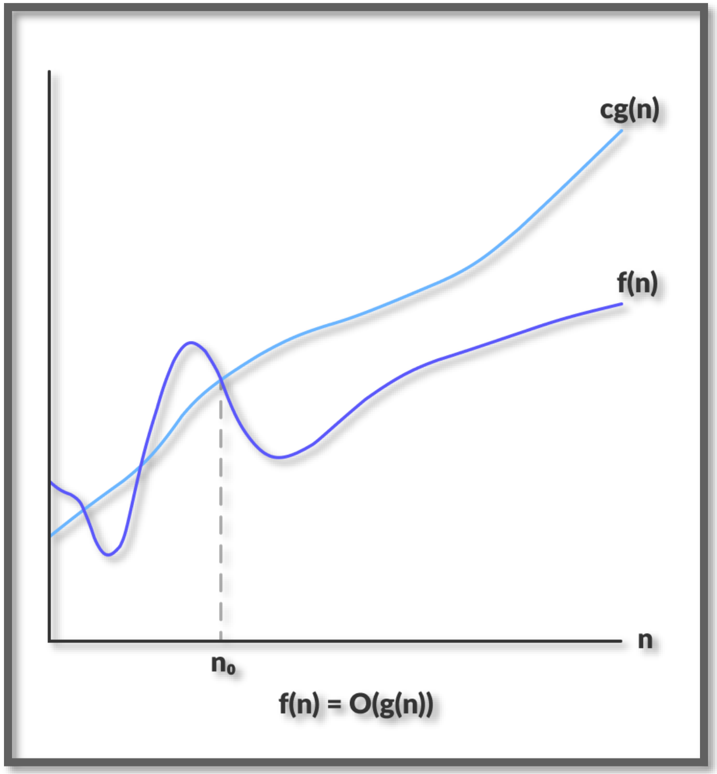
Approximation of T(n) is called ASYMPTOTIC COMPLEXITY

Asymptotic complexity studies the efficiency of an algorithm as the input size becomes large

Big-Oh or Big-O

f(n) = O(g(n)) *if there exist positive numbers* ***c & n_0***
such that ***f(n)<=cg(n)*** *for all $n >= n_0$*

g(n) is called the upper bound on f(n) OR
f(n) grows at the most as large as g(n)



How to choose c and N

We obtain the value of c & n by solving the inequality

$$2n^2 + 3n + 1 \leq cn^2$$

or equivalently

$$2 + \frac{3}{n} + \frac{1}{n^2} \leq c$$

Put value of n=1,2,3, in this equation to get value of c for that n

Because it is one inequality with two unknowns, different pairs of constants c and n for the same function g(n^2) can be determined.

For a fixed g, an infinite number of pairs of c's and n's can be identified.

Constants & Low order terms don't matter

- **"constants don't matter",**
  - $2n^3$ is $O(.001n^3)$.
    - Let $n_0 = 0$ and $c = 2/.001 = 2000$. Then clearly
    - $2n^3$ $2000(.001n^3) = 2n^3$, for all $n >= 0$.

- **"low order terms don't matter",**
- $T(n) = 3n^5 + 10n^4 - 4n^3 + n + 1$
    - The highest-order term is n^5, and we claim that $T(n)$ is $O(n^5)$.
    - To check the claim, let $n_0 = 1$ and let c be the sum of the positive coefficients.
    - **The terms with positive coefficients are those with exponents 5, 4, 1, and 0, whose coefficients are, respectively, 3, 10, 1, and 1.**
    - Thus, we let $c = 15$. We claim that for $n >= 1$, $3n^5 + 10n^4 - 4n^3 + n + 1 <= 3n^5 + 10n^5 + n^5 + n^5 = 15n^5$

LECTURE #3

ALGORITHM ANALYSIS of different CODEs

Nested For Loops

- ***For Loop is confusing as three statements are embedded in one***

```
sum = 0;
for(i=0; i<N; i++)
   for(j=0; j<N; j++)
       sum += arr[i][j];
```

Nested For Loops

- ***For is confusing as three statements are embedded in one***

`sum = 0;`	1
`for(i=0; i<N; i++)`	$c_1 N$
`    for(j=0; j<N; j++)`	$c_2 (N*N)$
`        sum += arr[i][j];`	$c_3 N^2$

$T(N) = 1 + c_1 N + c_2 (N*N) + c_3 N^2 = O(N^2)$

The total number of times a statement executes =
*outer loop times * inner loop times*

Nested For Loops

`for(i=0; i<N; i++)`	N+1
`        arr[i][i] =0;`	N
`sum = 0;`	1
`for(i=0; i<N; i++)`	N+1
`    for(j=0; j<N; j++)`	N(N+1)
`        sum += arr[i][j];`	N^2

$T(n) = O(N^2)$

Example 1 (arithmetic series)

Code	Cost
`i = 0;`	1
`while (i < n){`	N+1
`    sum = a[0];`	N
`    j = 1;`	N
`    while ( j <= i){`	??
`        sum += a[j];`	??
`        j++`	??
`    }`	
`    cout<<"sum of subarray 0 to "<< i <<" is "<<sum<<endl;`	N
`    i++`	
`}`	N

$$\sum_{i=1}^{n-1} 1 = N$$

as cost of this step is 1

j runs i times in each outer loop

i	j	# of time j loop runs
0		0
1	1	1
2		
3		
n-1		

Example 1 (arithmetic series)

- `i = 0;`
- `while (i < n){`
  - `sum = a[0];`
  - `j = 1;`
  - **`while ( j <= i){`**
    - `sum += a[j];`
    - `j++`
  - `}`
    - `cout<<"sum of subarray 0 to "<< i <<" is "<<sum<<endl;`
    - `i++`
- `}`

1
N+1
N
N
??
??
??
N
N

$$\sum_{i=1}^{n-1} 1 = N$$

as cost of this step is 1

j runs i times in each loop

i	j	# of time j loop runs
0		0
1	1	1
2	1,2	2
3		
n-1		

Example 1 (arithmetic series)

Code	Cost
`i = 0;`	**1**
`while (i < n){`	N+1
`– sum = a[0];`	N
`– j = 1;`	N
`– while ( j <= i){`	??
`• sum += a[j];`	??
`• j++`	??
`– }`	
`• cout<<"sum of subarray 0 to "<< i <<" is "<<sum<<endl;`	N
`• i++`	N
`}`	

$$\sum_{i=1}^{n-1} 1 = N$$

as cost of this step is 1

j runs i times in each loop

i	j	# of time j loop runs
0		0
1	1	1
2	1,2	2
3	1,2, 3	3
n-1		

Example 1 (arithmetic series)

- `i = 0;`
- `while (i < n){`
  - `sum = a[0];`
  - `j = 1;`
  - **`while ( j <= i){`**
    - `sum += a[j];`
    - `j++`
  - `}`
    - `cout<<"sum of subarray 0 to "<< i <<" is "<<sum<<endl;`
    - `i++`
- `}`

1
N+1
N
N
??
??
??
N
N

$$\sum_{i=1}^{n-1} 1 = N$$

as cost of this step is 1

j runs i times in each loop

i	j	# of time j loop runs
0		0
1	1	1
2	1,2	2
3	1,2, 3	3
n-1	**1,2, ..n-1**	**n-1**

$$\sum_{i=1}^{n-1} i$$

Example 1 (arithmetic series)

Code	Cost
• `i = 0;`	**1**
• `while (i < n){`	N+1
– `sum = a[0];`	N
– `j = 1;`	N
– **`while ( j <= i){`**	??
• `sum += a[j];`	??
• `j++`	??
– `}`	
• `cout<<"sum of subarray 0 to "<< i <<" is "<<sum<<endl;`	N
• `i++`	N
• `}`	

$$\sum_{i=1}^{n-1} 1 = N$$

as cost of this step is 1

j runs i times in each loop

i	j	# of time j loop runs
0		0
1	1	1
2	1,2	2
3	1,2, 3	3
n-1	**1,2, ..n-1**	**n-1**

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

This is arithmetic series

Sum up all to get

$$T(n) = O(n^2)$$

CUBE Example

	Cost	Times
`for (i=1; i<=n; i++)`	1	$\sum_{i=1}^{n-1} 1 = N$
`for (j=1; j<=i; j++)`	1	$\sum_{i=1}^{n-1} i$ ... *approx.no of times* ***this loop runs***
`for (k=1; k<=j; k++)`	1	$\sum_{i=1}^{n-1} \sum_{j=1}^{i} j$
`x=x+1;`	1	$\sum_{i=1}^{n-1} \sum_{j=1}^{i} j \approx \sum_{i=1}^{n-1} i(i+1)/2$

T(n) = sum up all

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \quad (\text{for n} >= 1)$$

➔ So, the growth-rate function for this algorithm is **O(n³)**

Multiply two matrices

- **int[][] multiply(int[][] A, int[][] B, int n)** {
  - int[][] product = new int[n][n];
  - for (int row = 0; row < n; row++) {
    - for (int col = 0; col < n; col++) {
      - int sum = 0;
      - for (int k = 0; k < n; k++)

        sum = sum + A[row][k] * B[k][col];
      - product[row][col] = sum;
    - }
  - }
    - return product;
- }

Examples

```
for (i=0;i<n;i=i+2)
      sum+=1;
```

Anything inside the loop will run approximately **n/2** times

Adding constant to i leads to O(N)

```
for (i=n;i>0;i=i-2)
      sum+=1;
```

Anything inside the loop will run approximately **n/2** times

This will still give O(N)

Let's see what multiplying with a constant leads to

```
for (i=1;i<n;i=i*2)
      sum+=1;
```

Anything inside the loop will run _____ times

EXAMPLE

```
counter = 0;
for (i=1; i<n; i=i*2)
    counter = counter+1;
```

i	**counter**
1	1
2	1+1=2
…	

EXAMPLE

```
counter = 0;
for (i=1;i<n;i=i*2)
    counter = counter+1;
```

i	**counter**
1	1
2	1+1=2
4	2+1=3
…	

EXAMPLE

```
counter = 0;
for (i=1;i<n;i=i*2)
    counter = counter+1;
```

i is increasing and it will continue to increase until i =n…

i	**counter**
1	1
2	1+1=2
4	2+1=3
8	3+1=4
…	

EXAMPLE

```
counter = 0;
for (i=1;i<n;i=i*2)
    counter = counter+1;
```

i	counter
1	1
2	1+1=2
4	2+1=3
8	3+1=4
…	
n/2	?
n	"exit loop"…

loop will not execute for i=n

i is increasing and it will continue to increase until i =n…

How many times i is incremented?

NOTE: i is incremented in powers of 2

2^0, 2^1 , 2^2, 2^3, … 2^m, where

$2^m = n$

Take log of both sides

$m = \log_2 n$

The value of counter is $\log_2 n$ at the end of the loop

EXAMPLE

```
counter = 0;
for (i=1;i<n;i=i*2)
    counter = counter+1;
```

i	counter
1	1
2	1+1=2
4	2+1=3
8	3+1=4
…	
n/2	$\log_2 n$
n	"exit loop"…

loop will not execute for i=n

i is increasing and it will continue to increase until i =n…

How many times i is incremented?

NOTE: i is incremented in powers of 2

$2^0, 2^1, 2^2, 2^3, \ldots 2^m$, where

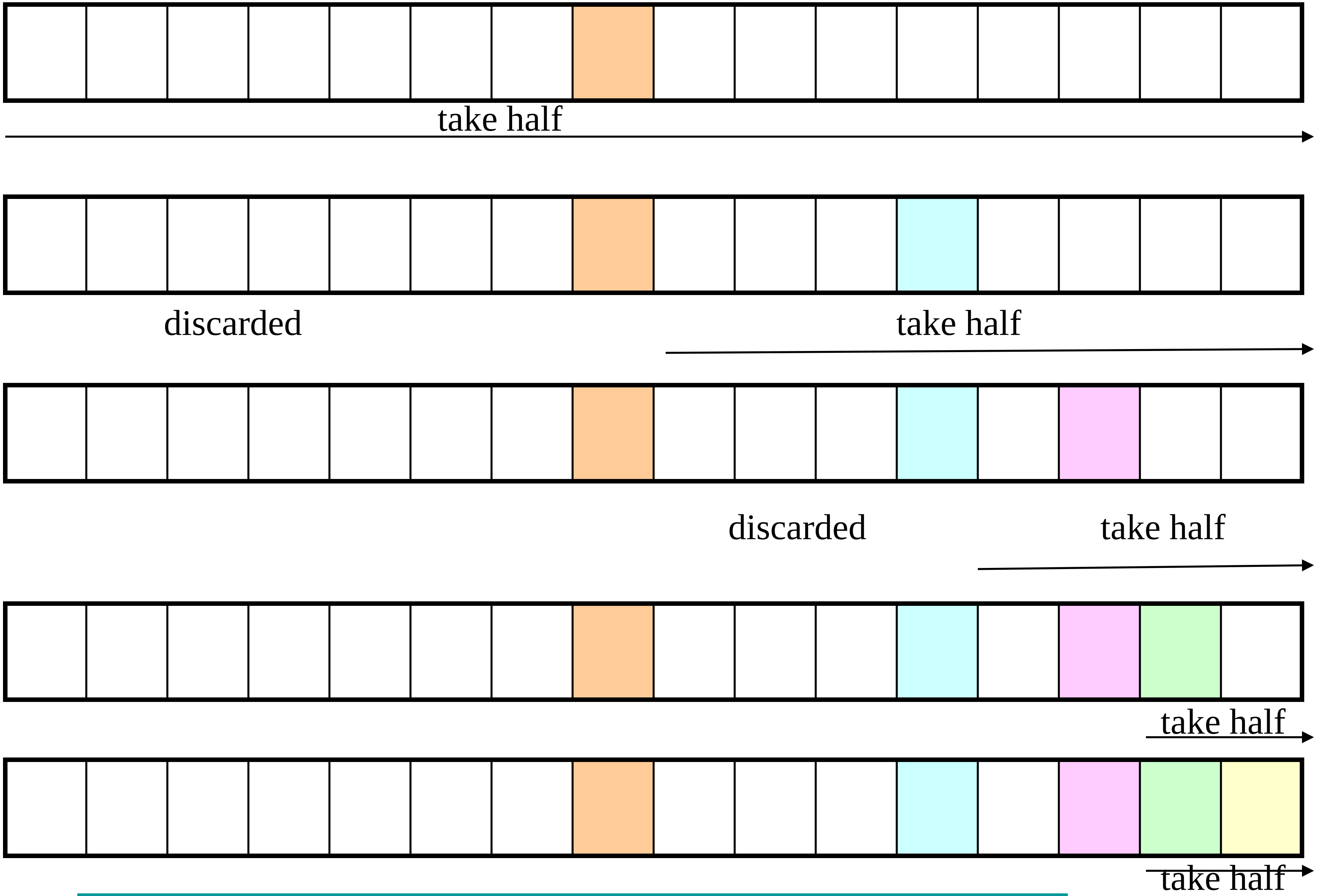
$2^m = n$

Take log of both sides

$m = \log_2 n$

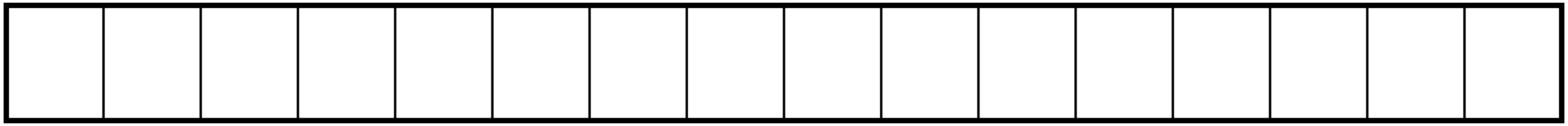
The value of counter is $\log_2 n$ at the end of the loop

EXAMPLE OF O(log n)…suppose you are coloring the table

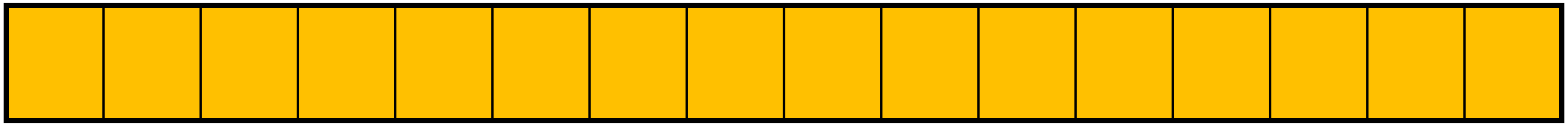


You didn't color all cells…..everytime you discarded half the cells

EXAMPLE OF O(n)…suppose you are coloring the table

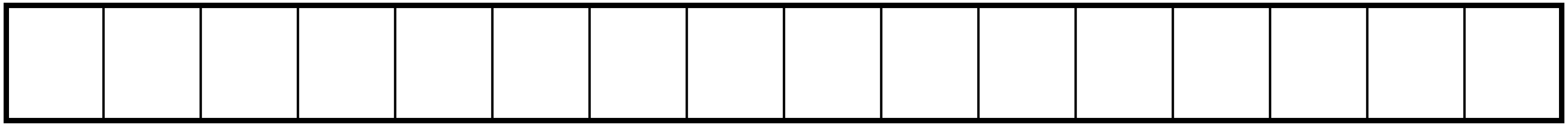


You will color all the cells one by one

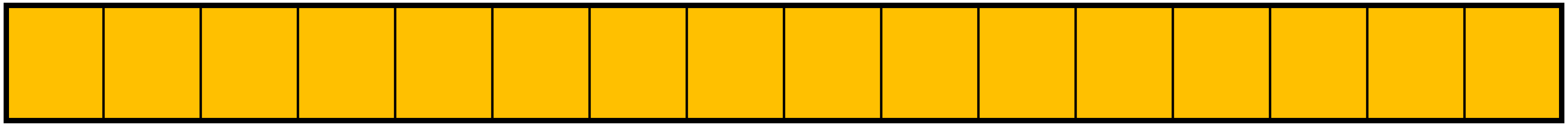


take half

EXAMPLE OF $O(n^2)$…suppose you are coloring the table



You will color each cell *n* times



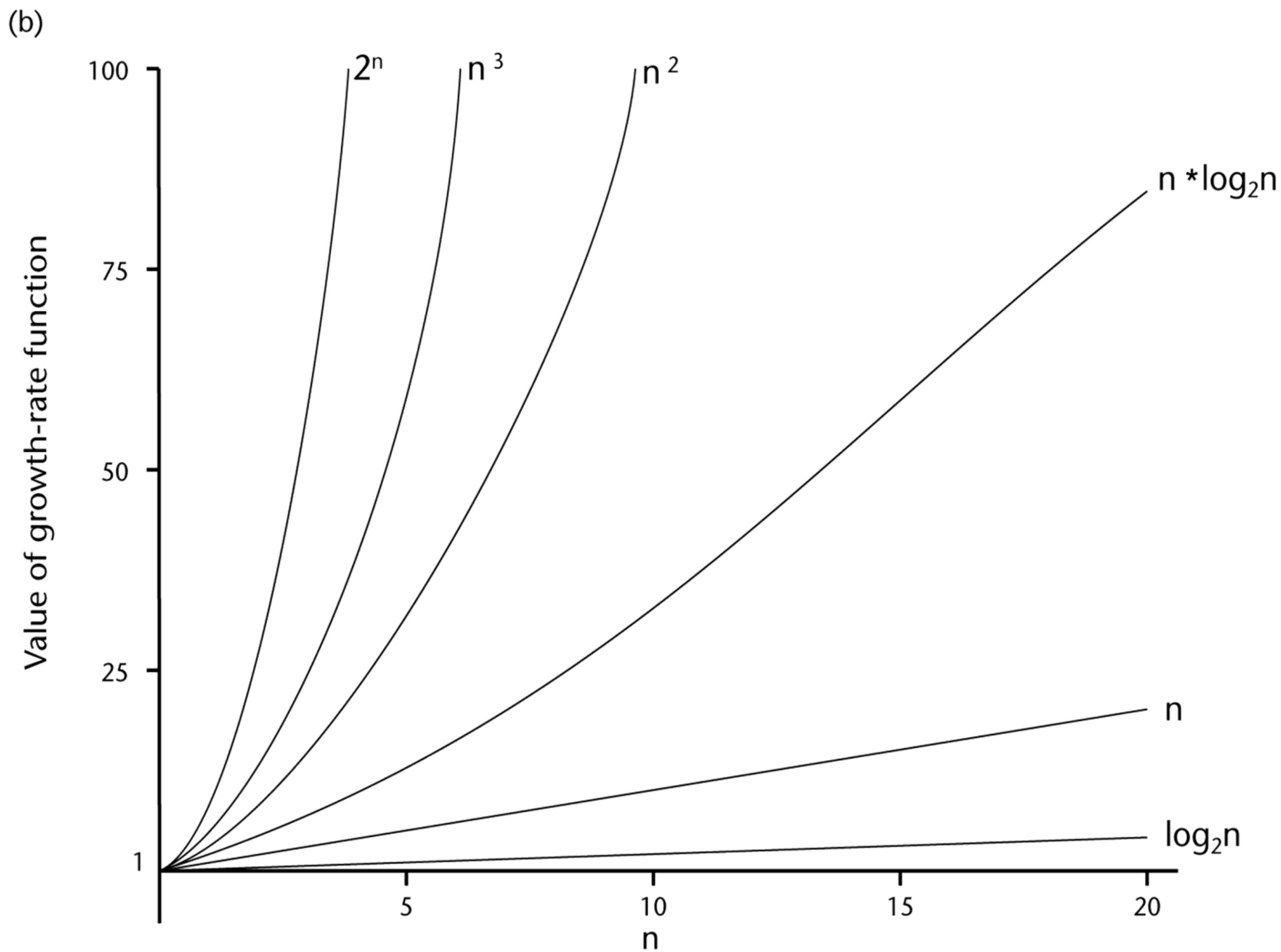
take half

EXAMPLE OF $O(2^n)$…suppose you are coloring the table

The number of times you color a cell is written there

1	2	4	8	16	32	64	128	256	512	1024	2048

A Comparison of Growth-Rate Functions (cont.)



How much better is $O(log_2 n)$?

n	$O(log_2 n)$
16	4
64	6
256	8
1024 (1KB)	10
16,384	14
131,072	17
262,144	18
524,288	19
1,048,576 (1MB)	20
1,073,741,824 (1GB)	30

Class	Complexity	Number of Operations and Execution Time (1 instr/μsec)		
n		10		
constant	$O(1)$	1	1 μsec	
logarithmic	$O(\lg n)$	3.32	3 μsec	
linear	$O(n)$	10	10 μsec	
$O(n \lg n)$	$O(n \lg n)$	33.2	33 μsec	
quadratic	$O(n^2)$	10^2	100 μsec	
cubic	$O(n^3)$	10^3	1 msec	
exponential	$O(2^n)$	1024	10 msec	

Class	Complexity	Number of Operations and Execution Time (1 instr/μsec)			
n		10		10^2	
constant	$O(1)$	1	1 μsec	1	1 μsec
logarithmic	$O(\lg n)$	3.32	3 μsec	6.64	7 μsec
linear	$O(n)$	10	10 μsec	10^2	100 μsec
$O(n \lg n)$	$O(n \lg n)$	33.2	33 μsec	664	664 μsec
quadratic	$O(n^2)$	10^2	100 μsec	10^4	10 msec
cubic	$O(n^3)$	10^3	1 msec	10^6	1 sec
exponential	$O(2^n)$	1024	10 msec	10^{30}	$3.17 * 10^{17}$ yrs

Class	Complexity	Number of Operations and Execution Time (1 instr/μsec)					
n		10		10^2		10^3	
constant	$O(1)$	1	1 μsec	1	1 μsec	1	1 μsec
logarithmic	$O(\lg n)$	3.32	3 μsec	6.64	7 μsec	9.97	10 μsec
linear	$O(n)$	10	10 μsec	10^2	100 μsec	10^3	1 msec
$O(n \lg n)$	$O(n \lg n)$	33.2	33 μsec	664	664 μsec	9970	10 msec
quadratic	$O(n^2)$	10^2	100 μsec	10^4	10 msec	10^6	1 sec
cubic	$O(n^3)$	10^3	1 msec	10^6	1 sec	10^9	16.7 min
exponential	$O(2^n)$	1024	10 msec	10^{30}	$3.17 * 10^{17}$ yrs	10^{301}	

Class	Complexity	Number of Operations and Execution Time (1 instr/μsec)					
n		10		10^2		10^3	
constant	$O(1)$	1	1 μsec	1	1 μsec	1	1 μsec
logarithmic	$O(\lg n)$	3.32	3 μsec	6.64	7 μsec	9.97	10 μsec
linear	$O(n)$	10	10 μsec	10^2	100 μsec	10^3	1 msec
$O(n \lg n)$	$O(n \lg n)$	33.2	33 μsec	664	664 μsec	9970	10 msec
quadratic	$O(n^2)$	10^2	100 μsec	10^4	10 msec	10^6	1 sec
cubic	$O(n^3)$	10^3	1 msec	10^6	1 sec	10^9	16.7 min
exponential	$O(2^n)$	1024	10 msec	10^{30}	$3.17 * 10^{17}$ yrs	10^{301}	

n		10^4					
constant	$O(1)$	1	1 μsec				
logarithmic	$O(\lg n)$	13.3	13 μsec				
linear	$O(n)$	10^4	10 msec				
$O(n \lg n)$	$O(n \lg n)$	$133 * 10^3$	133 msec				
quadratic	$O(n^2)$	10^8	1.7 min				
cubic	$O(n^3)$	10^{12}	11.6 days				
exponential	$O(2^n)$	10^{3010}					

Class	Complexity	Number of Operations and Execution Time (1 instr/μsec)					
n		10		10^2		10^3	
constant	$O(1)$	1	1 μsec	1	1 μsec	1	1 μsec
logarithmic	$O(\lg n)$	3.32	3 μsec	6.64	7 μsec	9.97	10 μsec
linear	$O(n)$	10	10 μsec	10^2	100 μsec	10^3	1 msec
$O(n \lg n)$	$O(n \lg n)$	33.2	33 μsec	664	664 μsec	9970	10 msec
quadratic	$O(n^2)$	10^2	100 μsec	10^4	10 msec	10^6	1 sec
cubic	$O(n^3)$	10^3	1 msec	10^6	1 sec	10^9	16.7 min
exponential	$O(2^n)$	1024	10 msec	10^{30}	$3.17 * 10^{17}$ yrs	10^{301}	

n		10^4		10^5			
constant	$O(1)$	1	1 μsec	1	1 μsec		
logarithmic	$O(\lg n)$	13.3	13 μsec	16.6	7 μsec		
linear	$O(n)$	10^4	10 msec	10^5	0.1 sec		
$O(n \lg n)$	$O(n \lg n)$	$133 * 10^3$	133 msec	$166 * 10^4$	1.6 sec		
quadratic	$O(n^2)$	10^8	1.7 min	10^{10}	16.7 min		
cubic	$O(n^3)$	10^{12}	11.6 days	10^{15}	31.7 yr		
exponential	$O(2^n)$	10^{3010}		10^{30103}			

Class	Complexity	Number of Operations and Execution Time (1 instr/μsec)					
n		10		10^2		10^3	
constant	$O(1)$	1	1 μsec	1	1 μsec	1	1 μsec
logarithmic	$O(\lg n)$	3.32	3 μsec	6.64	7 μsec	9.97	10 μsec
linear	$O(n)$	10	10 μsec	10^2	100 μsec	10^3	1 msec
$O(n \lg n)$	$O(n \lg n)$	33.2	33 μsec	664	664 μsec	9970	10 msec
quadratic	$O(n^2)$	10^2	100 μsec	10^4	10 msec	10^6	1 sec
cubic	$O(n^3)$	10^3	1 msec	10^6	1 sec	10^9	16.7 min
exponential	$O(2^n)$	1024	10 msec	10^{30}	$3.17 * 10^{17}$ yrs	10^{301}	

n		10^4		10^5		10^6	
constant	$O(1)$	1	1 μsec	1	1 μsec	1	1 μsec
logarithmic	$O(\lg n)$	13.3	13 μsec	16.6	7 μsec	19.93	20 μsec
linear	$O(n)$	10^4	10 msec	10^5	0.1 sec	10^6	1 sec
$O(n \lg n)$	$O(n \lg n)$	$133 * 10^3$	133 msec	$166 * 10^4$	1.6 sec	$199.3 * 10^5$	20 sec
quadratic	$O(n^2)$	10^8	1.7 min	10^{10}	16.7 min	10^{12}	11.6 days
cubic	$O(n^3)$	10^{12}	11.6 days	10^{15}	31.7 yr	10^{18}	31,709 yr
exponential	$O(2^n)$	10^{3010}		10^{30103}		10^{301030}	

BINARY SEARCH

int binarySearch(int arr[], int Size, int key)
{

- int low = 0, mid, high = Size-1;
- while (low <= high) {
    - mid = (low + high)/2;
    - if (key < arr[mid])
        - high = mid - 1;
    - else if (arr[mid] < key)
        - low = mid + 1;
    - else return mid; // success:
- }
- return -1;}

- If `key` is in the middle of the array, the loop executes only one time.
- How many times does the loop execute in the case where `key` is not in the array?
- ***k* is not in the array can be determined after lg *n* iterations of the loop.**

BINARY SEARCH

```
int binarySearch(int arr[], int Size, int key)
{
    – int low = 0, mid, high = Size-1;
    – while (low <= high) {
        • mid = (low + high)/2;
        • if (key < arr[mid])
            – high = mid - 1;
        • else if (arr[mid] < key)
            – low = mid + 1;
        • else return mid; // success:
    – }
    – return -1;
• }
```

- The algo first look at the middle element n/2

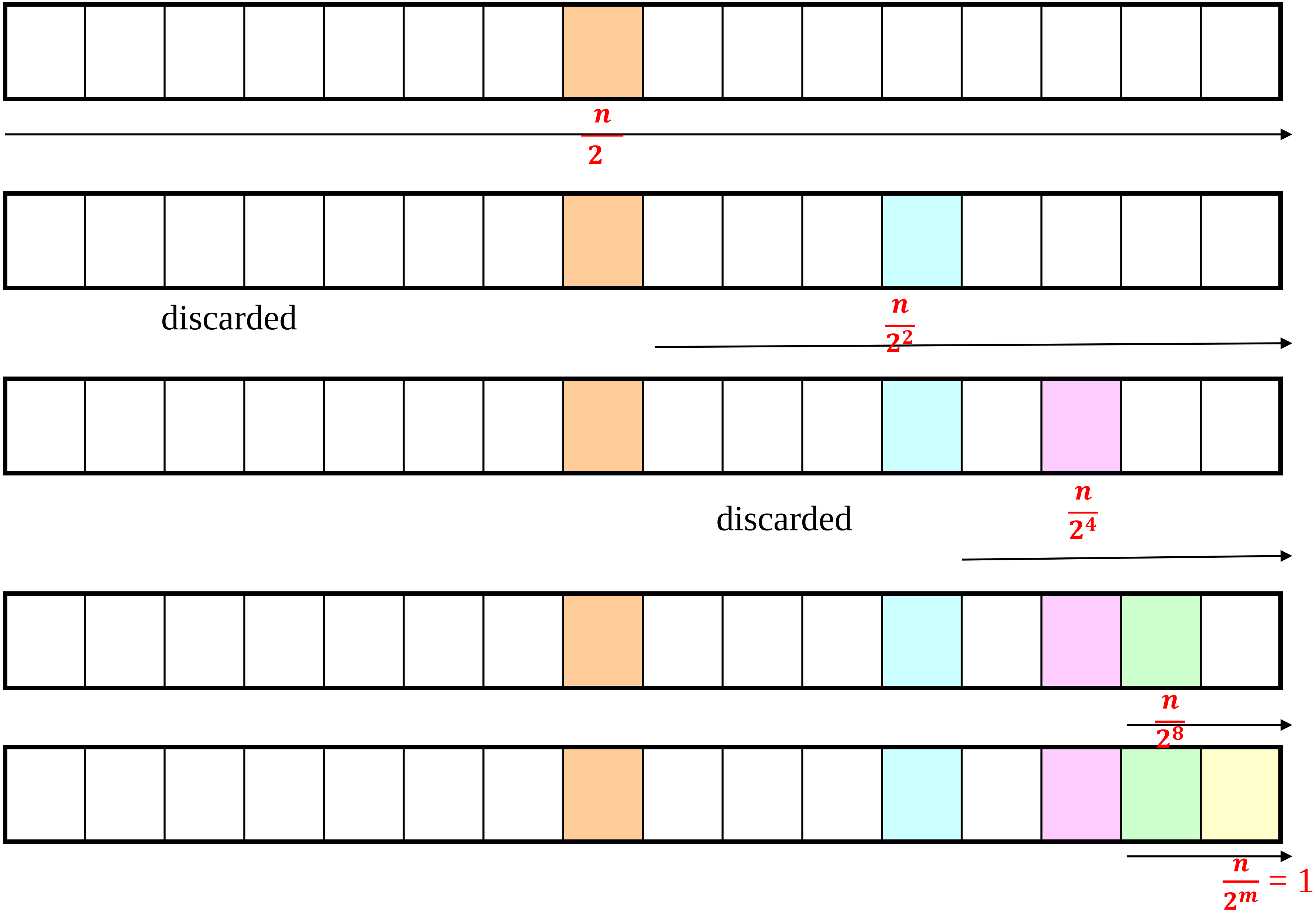
- **Then at the element at position** $\frac{n}{2^2}$ (half of the middle)
- **Then at at the element at position** $\frac{n}{2^4}$

- **It looks at elements at indices**
- $\frac{n}{2^0}\ \frac{n}{2^2}\ \frac{n}{2^4} \ldots \frac{n}{2^m}$

- **We wish to find the value of m**
- **But we know** $\frac{n}{2^m} = 1$
- $\mathbf{m = \log_2 n = lgn}$

So the fact that *k* is not in the array can be determined after lg *n* iterations of the loop.

EXAMPLE Binary search



Which g(n)- Inherent imprecision of the big-O

The big-O notation is **inherently imprecise** as there can be infinitely many functions ***g*** for a **given function *f*.**

- For example, the $f(n)=2\,n^2 + 3n+1$ is big-O not only of $\boldsymbol{n^2}$, but also of $\boldsymbol{n^3, \ldots, n^k}, \ldots$ for any $k \geq 2$.
- **To avoid this embarrassment of riches, the smallest function *g* is chosen, n^2 in this case.**

Real World Example:

- If a child, ask what comes after **class 1**
  - Answer could be class 2nd , 3rd ,4th, 5th , … but one should say **2nd**

PF -> OOP -> DS-> ALGO

Practise Question

- **Find complexity of**
  - Linear search
  - Bubble sort
  - Selection sort
  - Find Min
  - Factorial
  - Power(N,K)
  - N-bit Binary Counter …. *
- **Find c and no for**
  - $T(n) = 2^n + n^3$.
  - $T(n) = \lg n + n + 340$
  - $T(n) = n\lg n + 3$

Code Binary search and run it for different n

Code Linear Search and run it for different n

Code Bubble sort and run it for different n

n = 100, 1000, 10000, 100000 , ….
And note the time and report in the next class