EE 441 Data Structures

Lecture 3:

Algorithm Complexity

Algorithm

- An algorithm :
  - A computable set of steps to achieve a desired result.
  - precisely specified using an appropriate mathematical formalism--such as a programming language.
- Efficiency of an algorithm:
  - Less consumption of computing resources (execution time (CPU cycles), memory)
  - We will focus on time efficiency

Measuring Efficiency

- Two algorithms that accomplish the same task
  - *Which one is better???*
- You can run the algorithm and see the efficiency!!
- Benchmarking:
  - Run the program and measure runtime
  - Disadvantage:
    - Execution time depends on a number of different factors:
      - Programming language, compiler, operating system, computer architecture, input data.
    - No information about the fundamental nature of the program

Measuring Efficiency

- Given an algorithm, is it possible to determine how long it will take to run?
  – Input is unknown
  – Do not want to trace all possible execution paths
- For different inputs, is it possible to determine how an algorithm’s runtime changes?

Analysis of an Algorithm

- Predicting the resources that the algorithm requires
- Resources: memory, communication bandwidth, hardware but MOSTLY TIME
- In general the run time of a given algorithm grows by the size of the input
- Growth rate: How quickly the run time of an algorithm grows as a function of the problem input size
- Input size (N): number of items to be sorted, number of bits to represent the quantities etc.

Types of Analysis

- Worst case
  - Largest possible running time of algorithm on input of a given size.
  - Provides an upper bound on running time
  - An absolute guarantee that the algorithm would not run longer, no matter what the inputs are
- Best case
  - Provides a lower bound on running time
  - Input is the one for which the algorithm runs the fastest

$$\boxed{\textit{Lower Bound} \leq \textit{Running Time} \leq \textit{Upper Bound}}$$

Types of Analysis

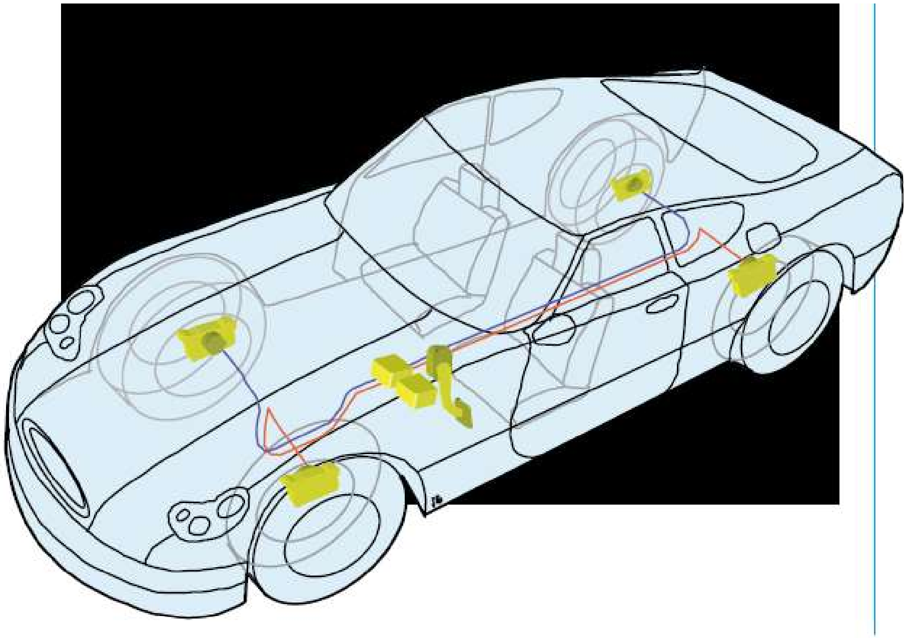
- Average Case:
  - Obtain bound on running time of algorithm on random input as a function of input size.
    - Hard (or impossible) to accurately model real instances by random distributions.
    - Algorithm tuned for a certain distribution may perform poorly on other inputs.

When does average case matter?

- Example: Design an algorithm that searches for a student in METU student database with certain properties (in third year, double major in physics)
- Worst case : No such student exists (rare). Algorithm searches the whole database and cannot find a match!
- Algorithm 1:
  - Average run time=1 sec
  - Worst case run time=8 sec
- Algorithm 2:
  - Average run time=4 sec
  - Worst case run time=5 sec
- WHICH ALGORITHM WILL YOU CHOOSE??

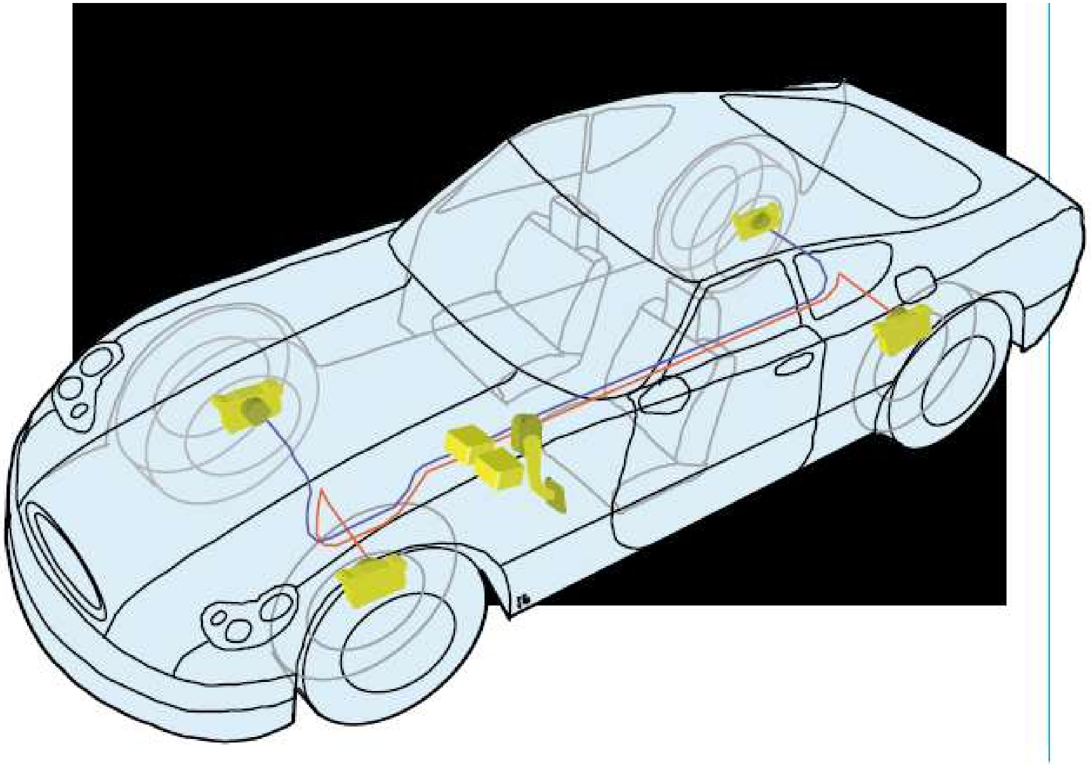
When does worst case matter?

- Example: Algorithm that computes the brake force in a brake by wire vehicle:
  - Brake is performed by a motor located at the wheels controlled by a computer
  - You are driving the car
  - You see the obstacle
  - The road is wet
  - You press on the brake



When does worst case matter?

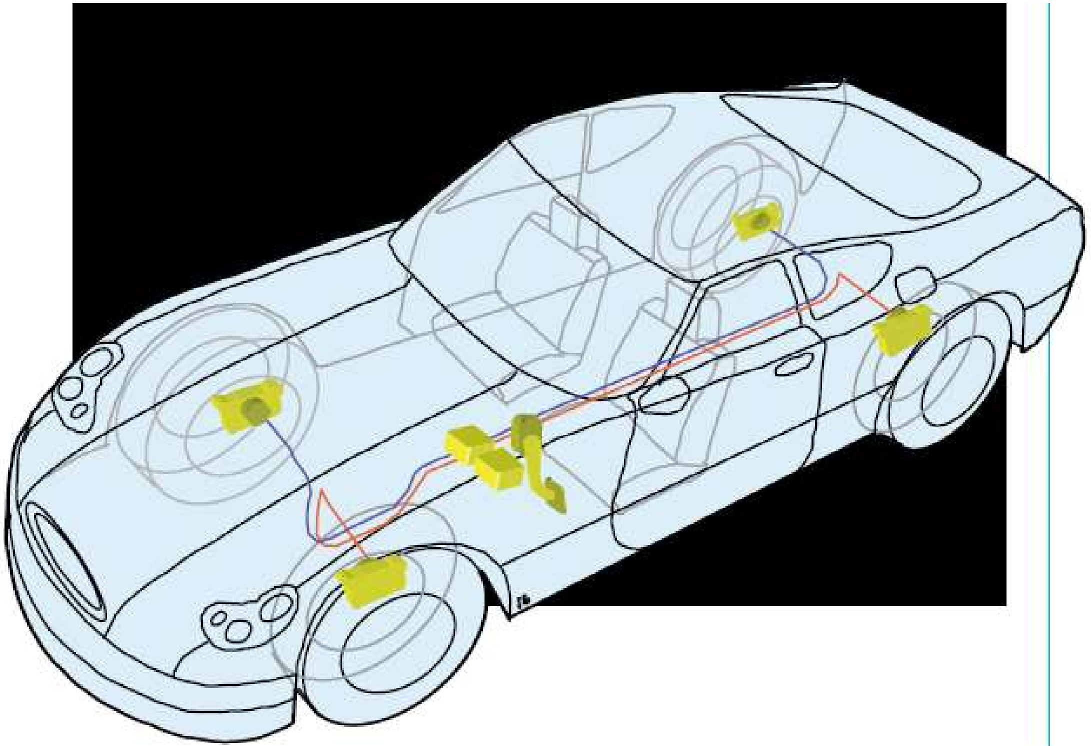
- An algorithm in the vehicle computer computes the actual braking force according to your braking force and road conditions.
- Sends the brake force value to the motor at the wheels



What is the design constraint of the algorithm???

When does worst case matter?

- Suppose: The algorithm has to generate a result in a maximum of 2msec such that you do not crash the object.
- The worst case run time should not exceed 2 msec
- You do not care about the average case!



Real time system that is life critical!

Measuring Efficiency

- Analysis:
  - Examine the program code
  - Assume each execution of statement *i* takes time t_i (constant)
  - Find how many times each statement is executed for a given input
  - Find worst cases
  - Some algorithms perform well for most cases but are very inefficient for few inputs: Average cases are important too!

Example

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

Checks including the last step where i>n for the first time

```
int sum (int n)
{
int result=0;                      → t1
for(int i=1;i<=n; i++)             → t2a t2b t2c
result+=i;                         → t3
return result;                     → t4
}
```

Time it takes to run:
T(n)=t1+t2a+(n+1)t2b+nt2c+nt3+t4

Some conclusions

- Running time T(n) depends on the problem size n.
- *Usually* T(n) is fixed for a certain n
- Question: What if the algorithm does some operations based on the outcome of a random variable?

Some conclusions

- We ignored the actual cost of each statement.
- We used t1 for time but we don't know how many nsec it takes to execute `int result=0` on Intel core i7 processor.
- We can simplify further → Just look at the TREND in time vs problem size rather than exact time

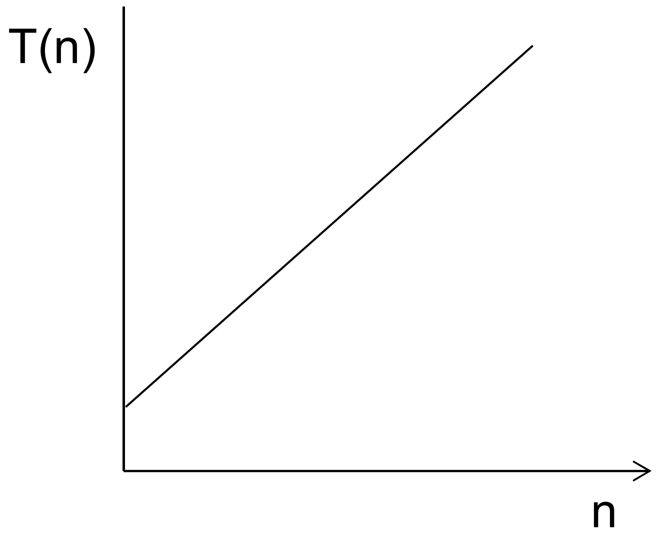
Rate of Growth

- Remember:

T(n)=t1+t2a+(n+1)t2b+nt2c+nt3+t4

T(n)= n(t2b+t2c+t3)+t1+t2a+t2b+t4

$T(n)=T_A n+T_B$



As $n \rightarrow \infty$:

T_B becomes insignificant with respect to nT_A

T_A does not change the shape of the curve

We are interested in the shape of the curve!!

Algorithm to solve a problem

- Problem:
  - An ordered array of N items
  - Find a desired item in the array
  - If the item exists in the array, return the index
  - Return -1 if no match is found
- There can be more than one solution→

Different algorithms

Algorithm 1: Sequential Search

- Idea:
  - Check all elements in the array one by one
  - from the beginning until:
    - The desired item is found ➔ Success
    - End of the array➔ no success

```
int SeqSearch(DataType list[ ],
  int n, DataType key)
{       // note DataType must be
  defined earlier
        // e.g., typedef int
  DataType;
        // or typedef float
  DataType; etc.
  for (int i=0; i<n; i++)
        if (list[i]==key)
              return i;
  return -1;
}
```

<u>worst case:</u>
n comparisons (operations) performed
<u>expected (average):</u>
n/2 comparisons
expected computation time α n

Algorithm 1: Sequential Search

- *expected computation time α n*
- e.g., if the algorithm takes 1 ms with 100 elements

  it takes ~5 ms with 500 elements

  ~200ms with 20000 elements etc.

Algorithm 2: Binary Search

- Idea:
  - Use a sorted array
  - Compare the element at the middle with the searched item
  - Decide which half of the array can contain the searched item

list[5]

5	17	36	37	45
0	1	2	3	4

low (initially) high (initially)

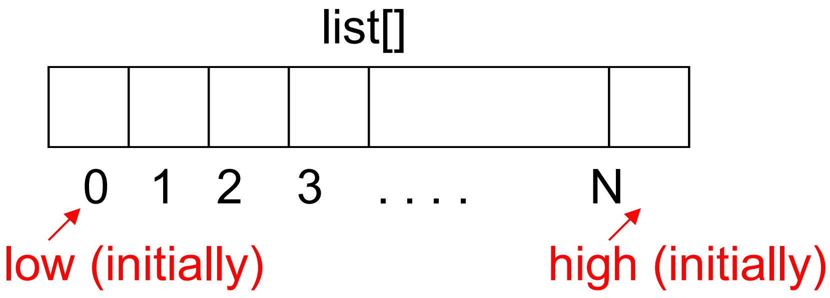
- Search for 37
- Middle is 36
- If 37 exists it has to be in the higher part of the array

5	17	36	37	45
0	1	2	3	4

low (next) high (next)

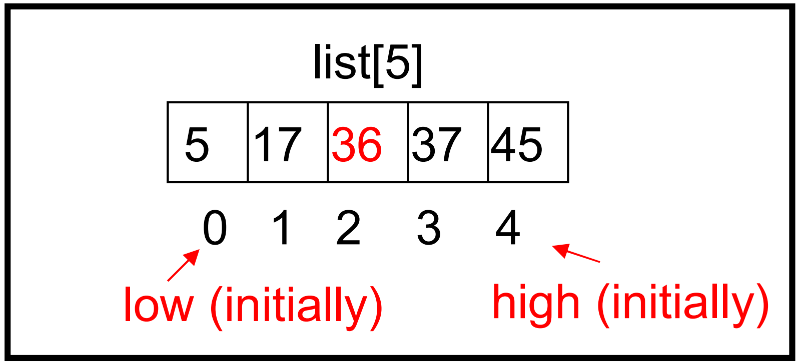
Algorithm 2: Binary Search

```
int BinarySearch(DataType list[], int low, int high,
DataType key)
{
        int mid;
        DataType midvalue;
        while (low<=high)
        {
                mid=(low+high)/2;    // note integer
division, middle of array
                midvalue=list[mid];
                if (key==midvalue) return mid;
                else if (key<midvalue) high=mid-1;
                        else low=mid+1;
        }
        return -1;
}
```

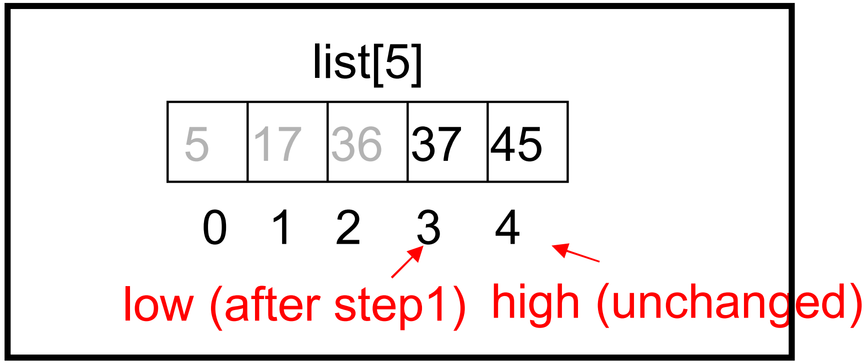


Binary Search

<u>e.g.</u> int list[5]={5,17,36,37,45};
low=0, high=4 key=44



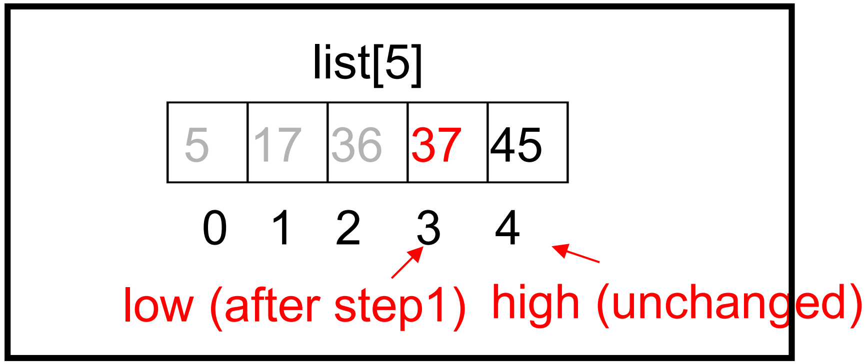
1) mid=(0+4)/2=2
    midvalue=list[2]=36
    key>midvalue
    low=mid+1=3

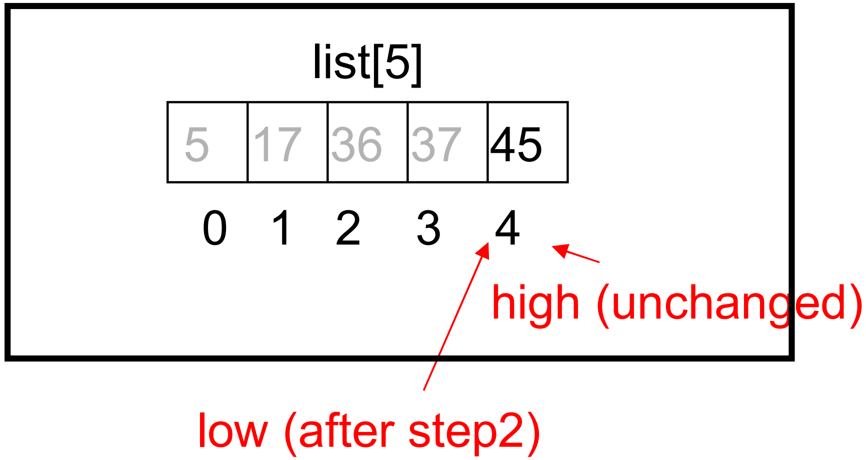


Binary Search

<u>e.g.</u> int list[5]={5,17,36,37,45};
low=0, high=4 key=44

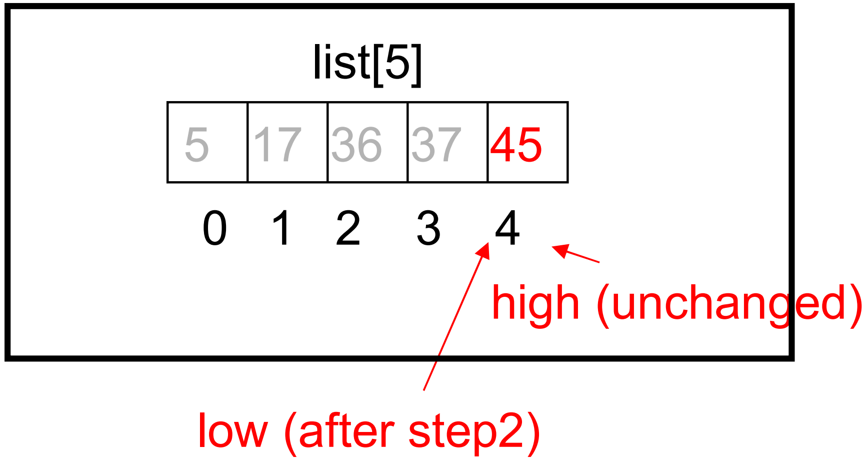
2) mid=(3+4)/2=3
midvalue=list[3]=37
key>midvalue
low=mid+1=4



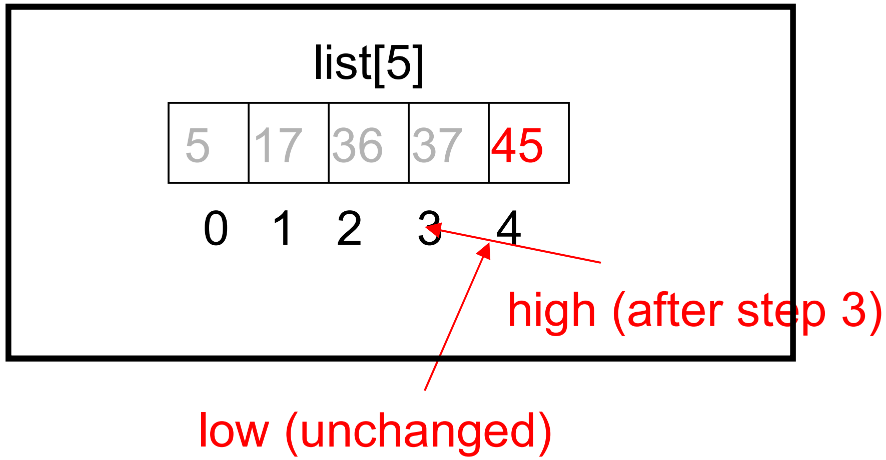


Binary Search

e.g. int list[5]={5,17,36,37,45};
low=0, high=4 key=44

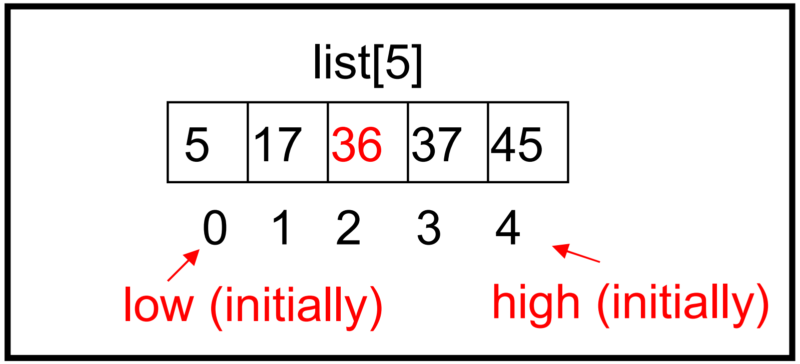


3) mid=(4+4)/2=4
midvalue=list[4]=45
key<midvalue
high=mid-1=3

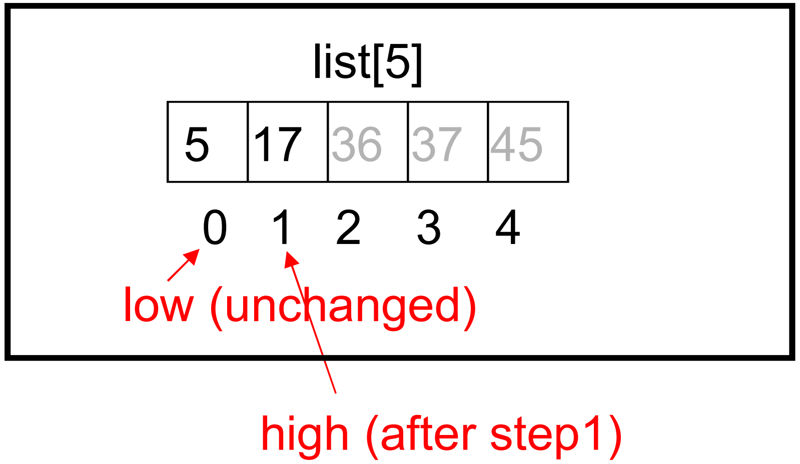


4) since high=3<low=4, exit the loop
return -1 (not found)

Binary Search



<u>e.g.</u>
int list[5]={5,17,36,37,45};
low=0, high=4 key=5
(the same example with different key)



1) mid=(0+4)/2=2
   midvalue=list[2]=36
   key<midvalue
   high=mid-1=1

Binary Search

<u>e.g.</u>
int list[5]={5,17,36,37,45};
low=0, high=4 key=5
(the same example with different key)

list[5]

5	17	36	37	45
0	1	2	3	4

low (unchanged)

high (after step1)

2) mid=(0+1)/2=0
midvalue=list[0]=5
key=midvalue
return 0 (found)

list[5]

5	17	36	37	45
0	1	2	3	4

Binary Search

- In the worst case, Binary Search makes $\lceil \log_2 n \rceil$ comparisons

(ceil) Smallest integer larger than or equal to

e.g.

n	$\log_2 n$
8	3
20	5
32	5
100	7
128	7
1000	10
1024	10
64000	16
65536	16

e.g. if Binary Search takes 1msec for 100 elements, it takes:

$t=k\lceil \log_2 n \rceil$

$1msec=k^*\lceil \log_2 100 \rceil$

$k=1/7$ msec/comparison

Hence, $t=(1/7)^*\lceil \log_2 n \rceil$

$t_{500}=(1/7)^*\lceil \log_2 500 \rceil=9/7\cong 1.29msec$

$t_{20000}=(1/7)^*\lceil \log_2 20000 \rceil=15/7\cong 2.1msec$

Computational Complexity

- Compares growth of two functions
- Independent of constant multipliers and lower-order effects
- Metrics
  - Big-O Notation: $O(\ )$
  - Big-Omega Notation: $\Omega(\ )$
  - Big-Theta Notation: $\Theta(\ )$
- Allows us to evaluate algorithms
- Has precise mathematical definition
- Used in a sense to put algorithms into families
- May often be determined by inspection of an algorithm

Definition: Big-O Notation

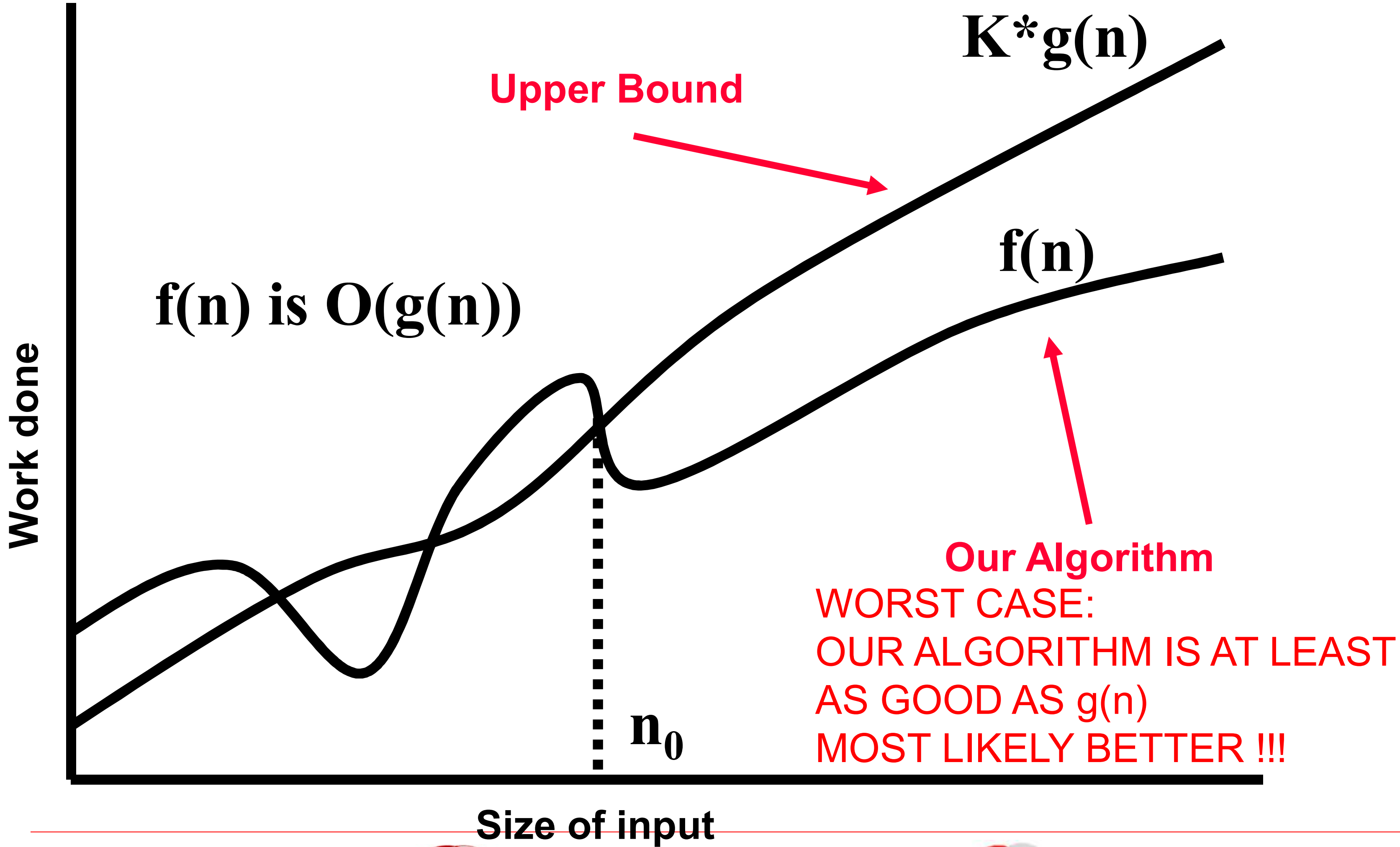
Function $f(n)$ is $O(g(n))$ if there exists a constant K and some n_0 such that

$$f(n) \le K^* g(n) \text{ for all } n \ge n_0$$

i.e., as $n \rightarrow \infty$, $f(n)$ is upper-bounded by a constant times $g(n)$.

- Usually, g(n) is selected among:
  - log n (note $\log_a n = k^* \log_b n$ for any $a, b \in \Re$)
  - n, n^k (polynomial)
  - k^n (exponential)

Big-O Notation

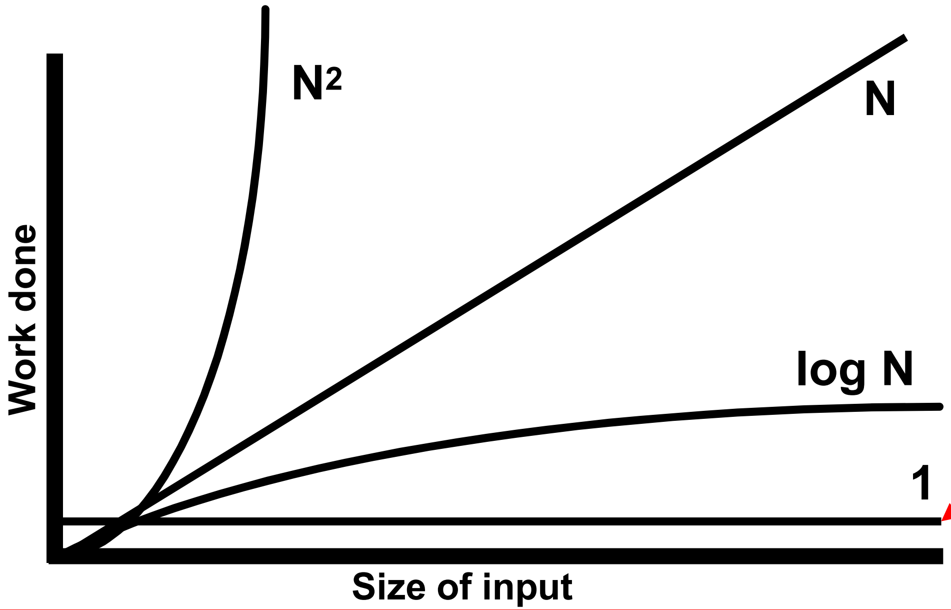


Comparing Two Algorithms

n	Seq. Search O(n)	Binary Search O(logn)
100	1 msec	1 msec
500	5 msec	1.3 msec
20000	200 msec	2.1 msec
...	...	...

Comparing Algorithms

- The O() of algorithms determined using the formal definition of O() notation:
  - Establishes the worst they perform
  - Helps compare and see which has "better" performance



O(1): Constant time
Work Done does not depend on the size of the input

Examples

e.g. $f(n)=n^2+250n+10^6$ is $O(n^2)$

because

$$f(n) \le n^2+n^2+n^2 \quad \text{for } n \ge 10^3$$
$$= 3n^2$$

K

n_0

e.g. $f(n)=2^n+10^{23}n+\sqrt{n}$ is $O(2^n)$

because

$10^{23}n<2^n$ for $n>n_0$ and $\sqrt{n} <2^n \; \forall n$

$f(n)\le 3*2^n$ for $n>n_0$

K

No Uniqueness

- There is no unique set of values for n_0 and K in proving the asymptotic bounds
- Prove that $100n + 5 = O(n^2)$
  - $100n + 5 \leq 100n + n = 101n \leq 101n^2$

    for all $n \geq 5$

    $n_0 = 5$ and $K = 101$ is a solution
  - $100n + 5 \leq 100n + 5n = 105n \leq 105n^2$

    for all $n \geq 1$

    $n_0 = 1$ and $K = 105$ is also a solution

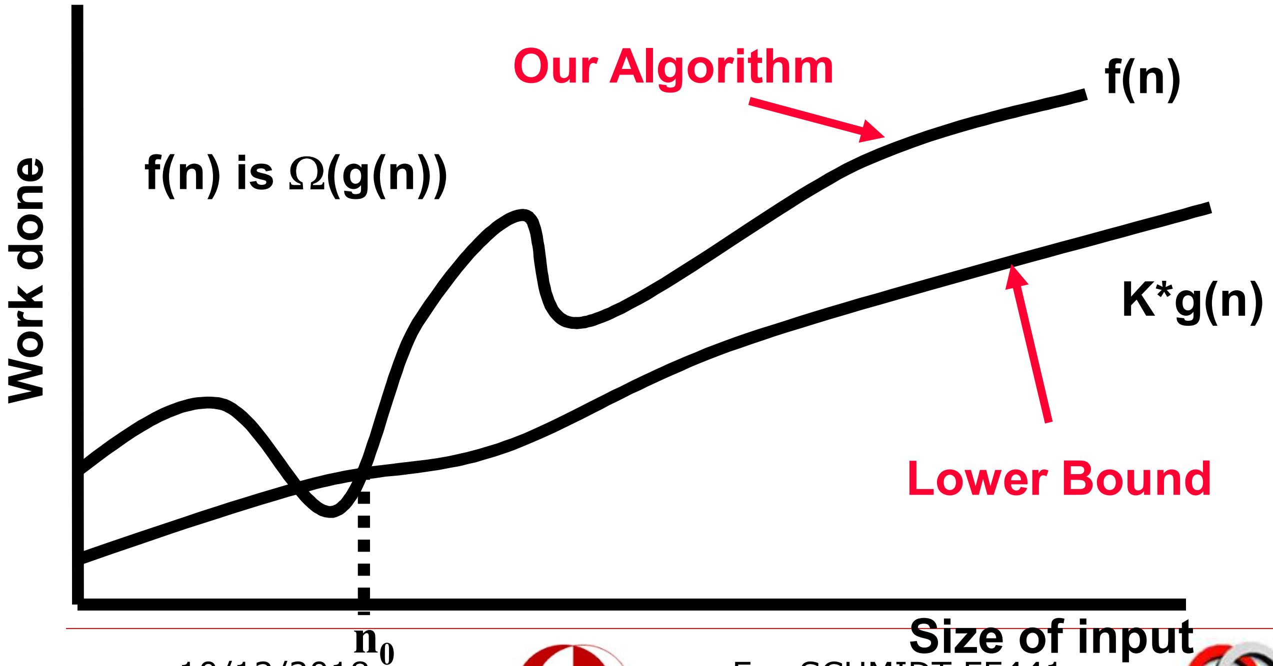
Must find **SOME** constants K and n_0 that satisfy the asymptotic notation relation

Big-Omega Notation

Function f(n) is $\Omega(g(n))$ if there exists a constant K and some n_0 such that

$$K^*g(n) \leq f(n) \text{ for all } n \geq n_0$$

i.e., as $n \to \infty$, f(n) is lower-bounded by a constant times g(n).



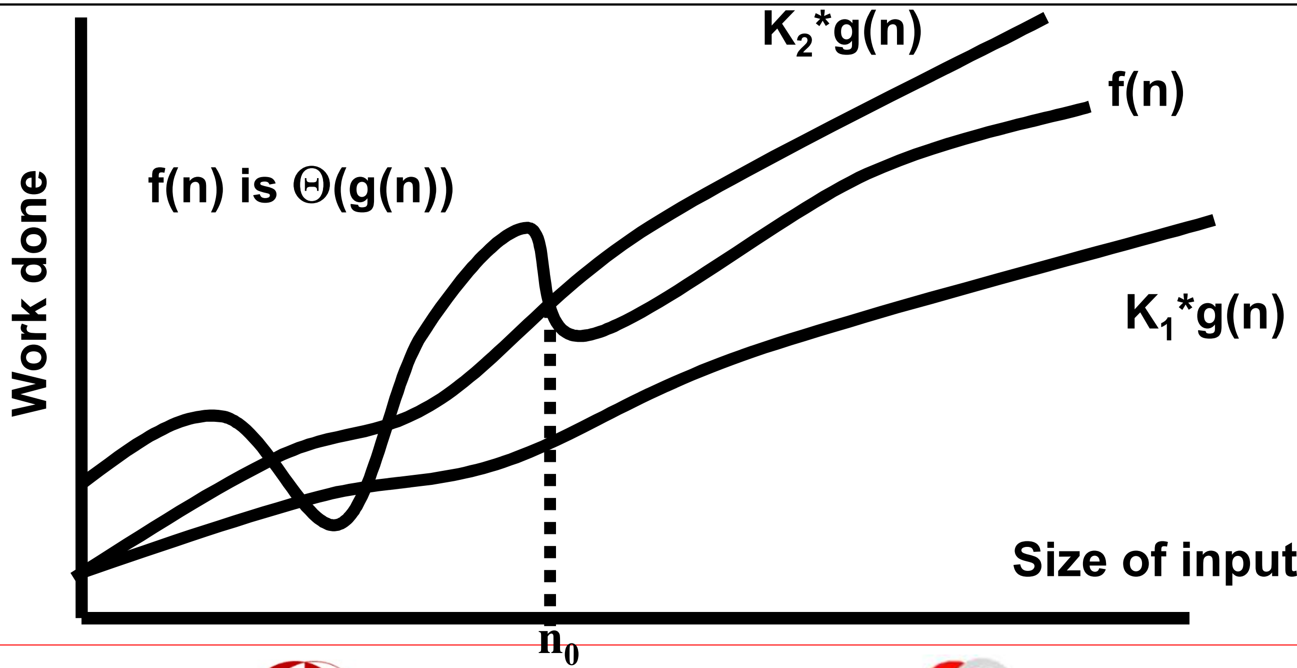
BEST CASE:
OUR ALGORITHM IS AT MOST AS GOOD AS g(n)
MOST LIKELY WORSE!!!

Big-Theta Notation

Function f(n) is Θ(g(n)) if there exist constants K_1 and K_2 and some n_0 such that

$$K_1*g(n) \le f(n) \le K_2*g(n) \text{ for all } n \ge n_0$$

i.e., as $n \rightarrow \infty$, f(n) is upper and lower bounded by some constants times g(n).



Asymptotic Notation

- O notation: asymptotic "less than":
  - f(n) is O(g(n)) implies: f(n) "≤" g(n)
- Ω notation: asymptotic "greater than":
  - f(n) is Ω (g(n)) implies: f(n) "≥" g(n)
- Θ notation: asymptotic "equality": TIGHT BOUND
  - f(n)is Θ (g(n)) implies: f(n) "=" g(n)

Theorem

- *Theorem:*

$$f(n) = \Theta(g(n)) \Leftrightarrow f = O(g(n)) \text{ and } f = \Omega(g(n))$$

f(n) is $\Theta(g(n))$ if f(n) is both $O(g(n))$ and $\Omega(g(n))$

Properties

- Transitivity:
  - $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$
  - Same for O and Ω
  - Example:
    - $f(n)=\log(n)$, $g(n)=n^2$, $h(n)=n!$
    - Given: $f(n)$ is $O(g(n))$.
    - $g(n)$ is $O(h(n)) \Rightarrow f(n)$ is $O(h(n))=O(n!)$

Properties

- Additivity:
  - $f(n) = \Theta(h(n))$ and $g(n) = \Theta(h(n))$ then $f(n) + g(n) = \Theta(h(n))$
  - Same for O and Ω
- Reflexivity:
  - $f(n) = \Theta(f(n))$
  - Same for O and Ω
- Symmetry:
  - $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$
- Transpose symmetry:
  - $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$

Common Asymptotic Bounds

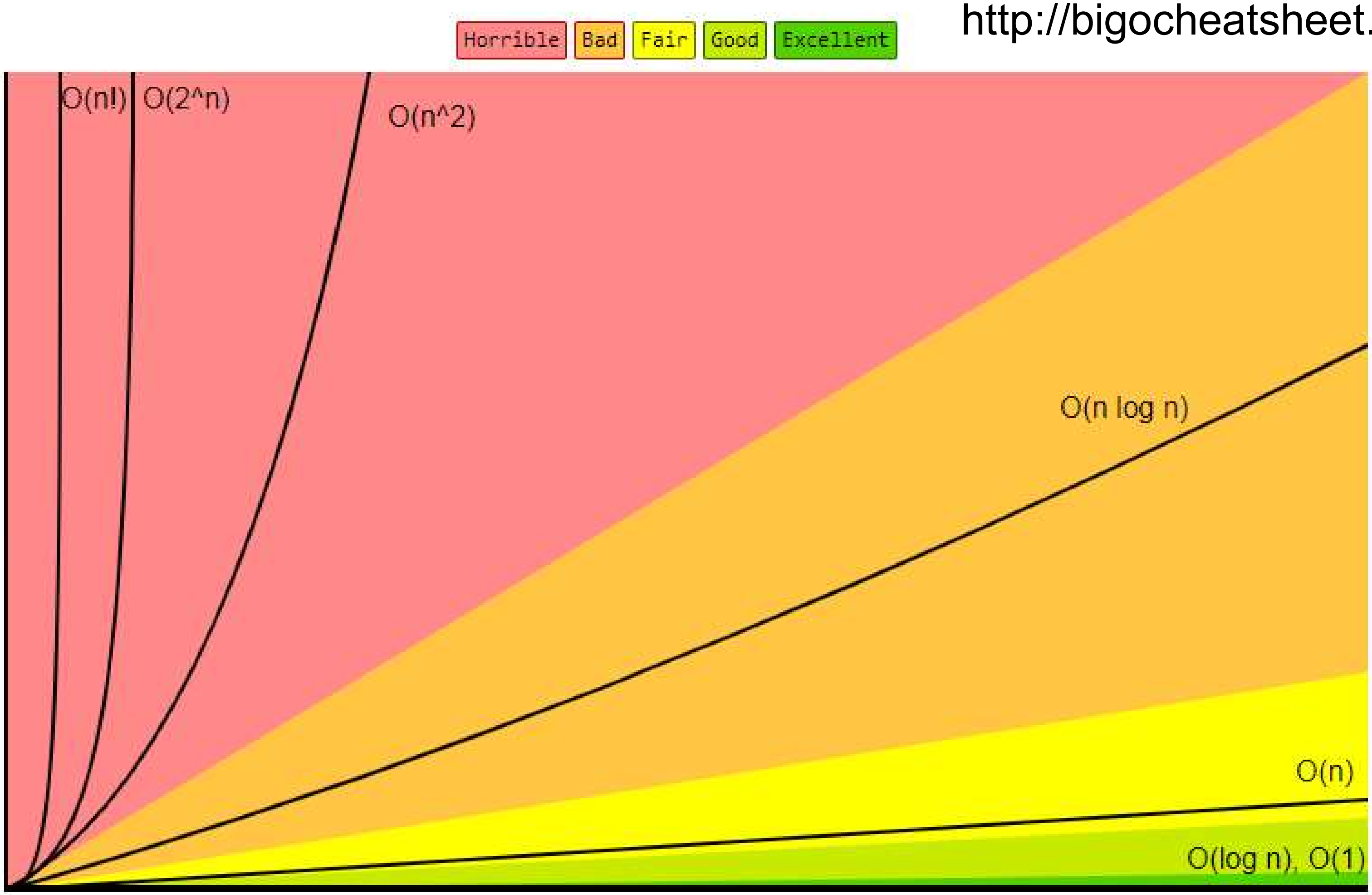
- Polynomials. $a_0 + a_1 n + \dots + a_d n^d$ is $\Theta(n^d)$ if $a_d > 0$.
- *Polynomial time.* Running time is $O(n^d)$ for some constant d that is independent of the input size n.

Common Asymptotic Bounds

- Logarithms. $O(\log_a n) = O(\log_b n)$ for any constants a, b > 0.
  - So, you can state logarithms without base
- For every $x > 0$, $\log n = O(n^x)$.
  - every polynomial grows faster than every log
- Exponentials. For every $r > 1$ and every $d > 0$, $n^d = O(r^n)$.
  - every exponential grows faster than every polynomial

Compare

http://bigocheatsheet.com/



Example

- $\mathrm{f(n)} = \frac{1}{2}n^2 + 3n$ is $\Theta(n^2)$
- We want K_1,K_2 and n_0 such that

$$K_1 n^2 \leq \frac{1}{2}n^2 + 3n \leq K_2 n^2$$

- Divide all expression by n^2

$$K_1 \leq \frac{1}{2} + 3\frac{1}{n} \leq K_2$$

Holds for $n > 1, K_1 = 0.5$ and $K_2 = 3.5$

Example

- myfunc1:$\Theta(n)$
- myfunc2:$\Theta(n^2)$
- myfunc3:$O(1)$

```
int RandFunc(int n, int seed)
{int x= Rand(seed);
for(int i=0;i<n;i++)
{if(x%2==0)myfunc1+myfunc3;
else myfunc2+myfunc1;}
}
```

- If x is always even: n times execute `myfunc1+myfunc3`
  $\mathrm{n}\big(\Theta(n) + O(1)\big) = \big(\Theta(n^2) + \Theta(n)\big) = \Theta(n^2)$
- If x is always odd: n times execute `myfunc2+myfunc1`
  $\mathrm{n}\big(\Theta(n^2) + \Theta(n)\big) = \Theta(n^3) + \Theta(n^2) = \Theta(n^3)$

Example

- If x is always even: n times execute `myfunc1+myfunc3`
  $\mathrm{n}\big(\Theta(n) + O(1)\big) = \big(\Theta(n^2) + \Theta(n)\big) = \Theta(n^2)$
- If x is always odd: n times execute `myfunc2+myfunc1`
  $\mathrm{n}\big(\Theta(n^2) + \Theta(n)\big) = \Theta(n^3) + \Theta(n^2) = \Theta(n^3)$
- Worst case:
  - x is always odd: $\Theta(n^3) \Rightarrow O(n^3)$
- Best case:
  - x is always even: $\Theta(n^2) \Rightarrow \Omega(n^2)$
- There is no Θ for `RandFunc`

Example

$$\sum_{i=1}^{n} a_i x^i$$

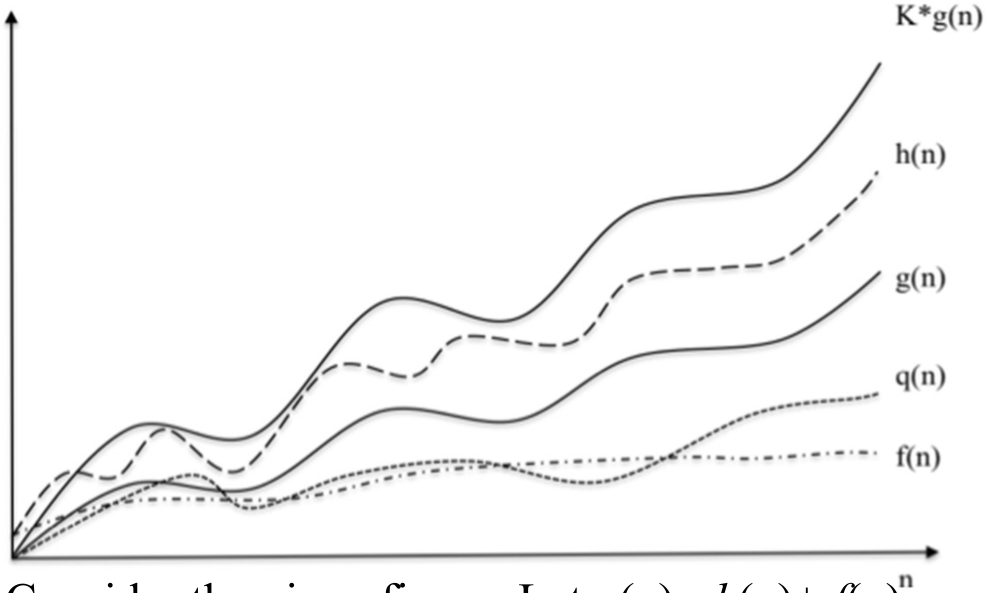
```
int Power (int a[], int n, int x)
{int xpower=1;  t1
result=a[0]*xpower;    t2
for(int i=1;i<=n;i++)  t3a t3b t3c
{xpower=x*xpower;      t4
result+=a[i]*xpower;}  t5
return result;         t6
}
```

T(n)=t1 + t2 + t3a + (n+1)t3b + n(t3c+t4+t5) + t6

T(n)=TA+nTB

$O(n), \Theta(n), \Omega(n)$

Example



Consider the given figure. Let $p(n) = h(n) + f(n)$

Find the TIGHTEST O(.) ,Ω(.) and Θ(.) complexities of $h(n)$, $q(n)$ and $p(n)$ expressed in terms of $f(n)$ and $g(n)$.

Solution:

$h(n)$: $\Theta(g(n))$

$q(n)$: $O(g(n))$, $\Omega(f(n))$

$p(n)$: $\Theta(g(n))$

EE 441 Data Structures

Lecture 3:

Algorithm Complexity