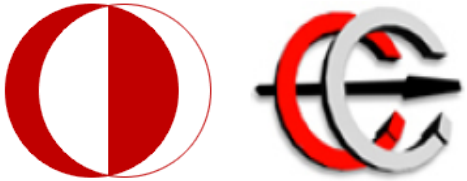


EE 441 Data Structures

Lecture 4: Algorithm Complexity

Algorithm

- An algorithm is
  - a computable set of steps to achieve a desired result
  - precisely specified using an appropriate mathematical formalism

    → For example programming language
- Examples for Algorithms
  - The algorithm of google maps to find the shortest driving route between two locations on a map
  - Data compression algorithms
  - Search algorithms
  - Object detection algorithms
  - Control algorithms
  - Filtering algorithms
  - .....

Efficiency of an Algorithm

- Objective: Less consumption of computing resources
  - Execution time (CPU cycles)
  - Memory
  - FPGA LUTs
  - Network bandwidth

  → We will mostly focus on time efficiency
- Question: Two algorithms that accomplish the same task
  - Which one is better???
- Simple idea: Benchmarking
  - Run the program and measure runtime
- Disadvantage: Execution time depends on different factors
  - Programming language, compiler, operating system, computer architecture
  - Input data

  → No information about the fundamental nature of the program

Measuring Efficiency

- Question 1: 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

  → Question is difficult to answer
- Question 2: For different inputs, is it possible to determine how an algorithm's runtime changes?

  → We will analyze algorithms to answer this question (partially)

Analysis of an Algorithm

- Predicting the resources that the algorithm requires
  - Memory
  - Communication bandwidth
  - Hardware
  - MOSTLY TIME
- General fact: Run time of a given algorithm grows with the input size
- 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
  - Serves as 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
  - Determined by the input for which the algorithm runs the fastest
- 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

Discussion

- General Case

$$Lower\ Bound \leq Running\ Time \leq Upper\ Bound$$

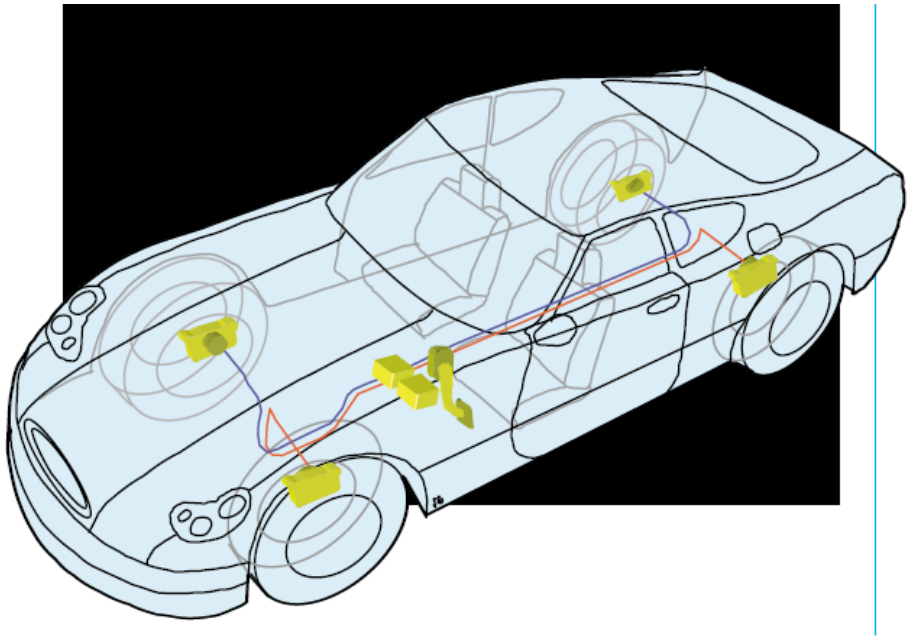
- Example: Design an algorithm that searches for a student in METU student database with certain properties (third year, double major in physics)
  - Worst case: No such student exists (rare)

    → 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??**

Discussion

- 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
  - An algorithm on an electronic control unit (ECU) computes the actual braking force according to your braking force and road conditions
  - Sends the brake force value to the motor at the wheels

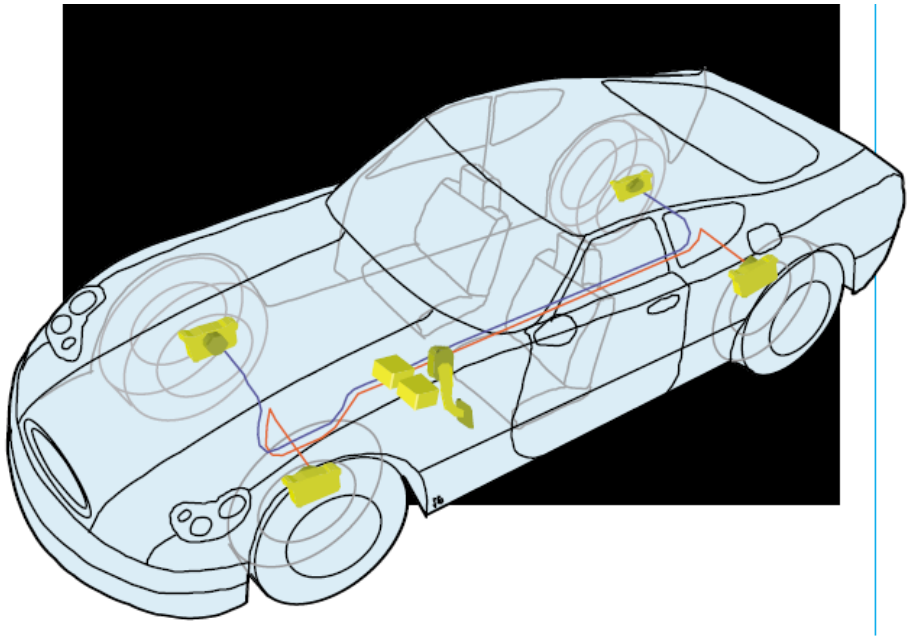


What is the design constraint of the algorithm???

When does the worst case matter?

- Suppose: The algorithm has t generate a result in a maximum of 2 msec such that you do not crash the object
- The worst case run time should not exceed 2 msec

  → In that case, 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
  - Note: Some algorithms perform well for most cases but are very inefficient for few inputs

    → Average cases are important too!
- Example

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

```
int sum (int n){
  int result=0;
  for(int i=1;i<=n; i++)
    result+=i;
  return result;
}
```

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

→ $t1$
→ $t2a$ $t2b$ $t2c$
→ $t3$
→ $t4$

Time it takes to run: $T(n) = t1 + t2a + (n+1) \cdot t2b + n \cdot t2c + n \cdot t3 + t4$

Some Conclusions

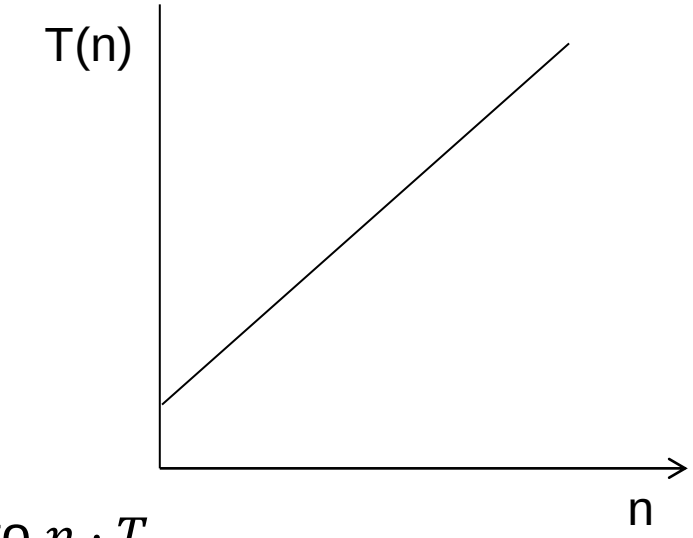
- Running time $T(n)$ depends on the problem size n
- Usually $T(n)$ is fixed for a certain n
- Question: What if the algorithm does some operations based on the outcome of a random variable?
- Issue: We ignored the actual cost of each statement
- Issue: We used $t1$ for time but we don't know how many nsec it takes for example to execute int result = 0 on an Intel core i7 processor
- We will simplify further

  → Just look at the TREND in time vs problem size rather than exact time

Rate of Growth

- Remember

$$T(n) = t1 + t2a + (n+1) \cdot t2b + n \cdot t2c + n \cdot t3 + t4$$
$$T(n) = n \cdot (t2b + t2c + t3) + t1 + t2a + t2b + t4$$
$$T(n) = n \cdot T_A + TB$$



- Important observation: As $n \to \infty$
  - T_B becomes insignificant with respect to $n \cdot T_A$
  - T_A does not change the shape of the curve

$\to$ We are interested in the shape of the curve!!

Algorithm to solve a problem

- Example 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
- Fact: There can be more than one solution

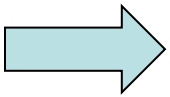
  $\rightarrow$ Different algorithms should be studied and compared

Algorithm 1: Sequential Search

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

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

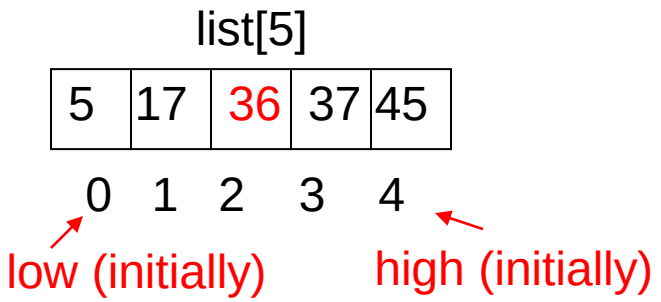
- Analysis
  - Worst case: n comparisons (operations) performed
  - Expected (average): $n/2$ comparisons
  - Expected computation time $\propto n$
- Meaning
  - Assume the algorithm takes 1 ms for 100 elements



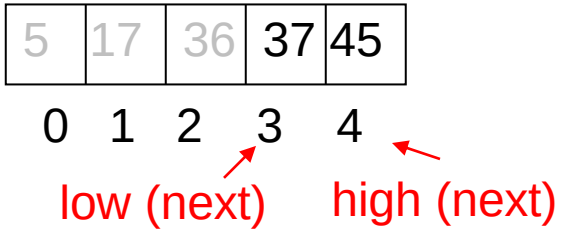
~5 ms with 500 elements
~200ms with 20000 elements etc.

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 must contain the searched item if it exists in the array

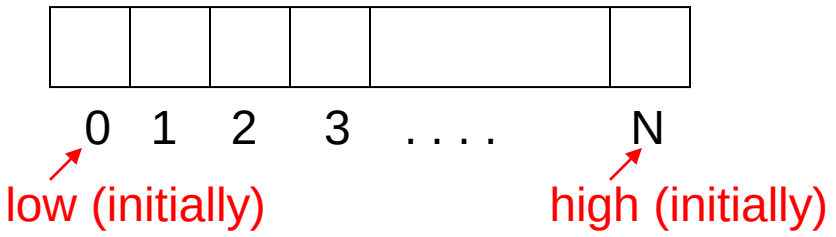


- Search for 37
- Middle is 36
- If 37 exists it has to be in the higher part of the array



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

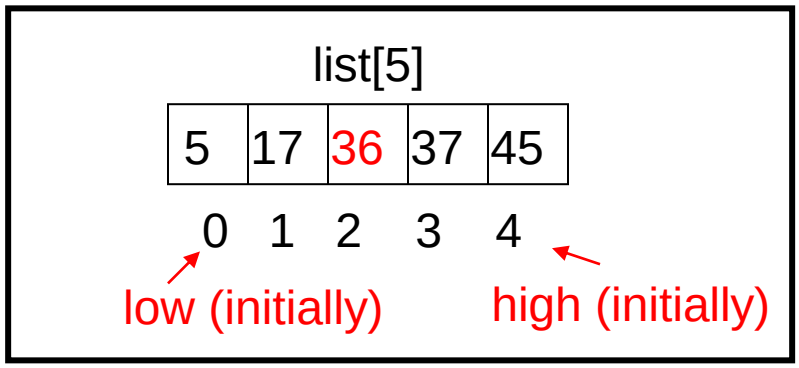
Example

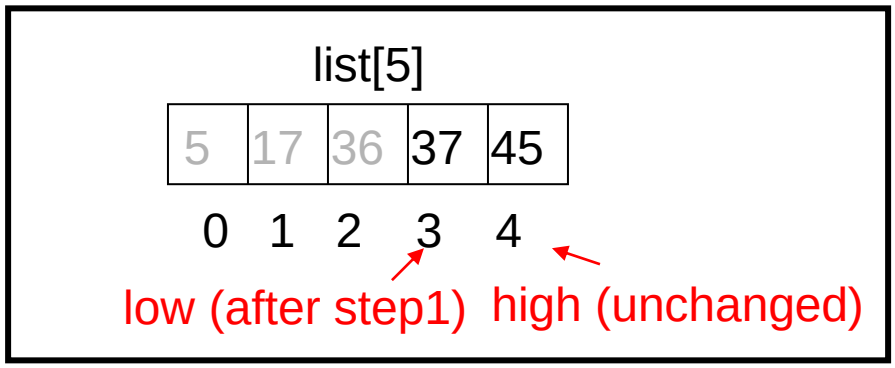
```
int list[5]={5,17,36,37,45};
```

low=0, high=4 key=44

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

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





Binary Search

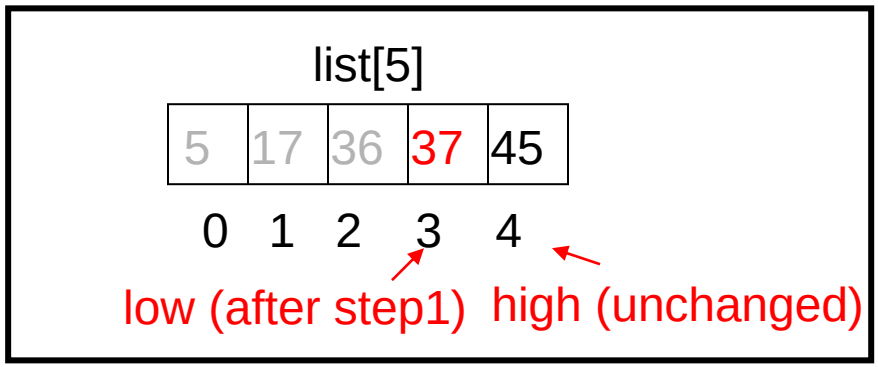
Example

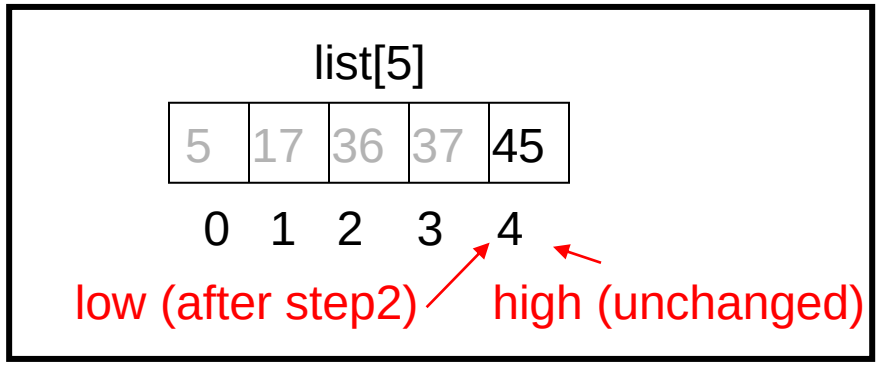
```
int list[5]={5,17,36,37,45};
```

low=0, high=4 key=44

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

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





Binary Search

Example

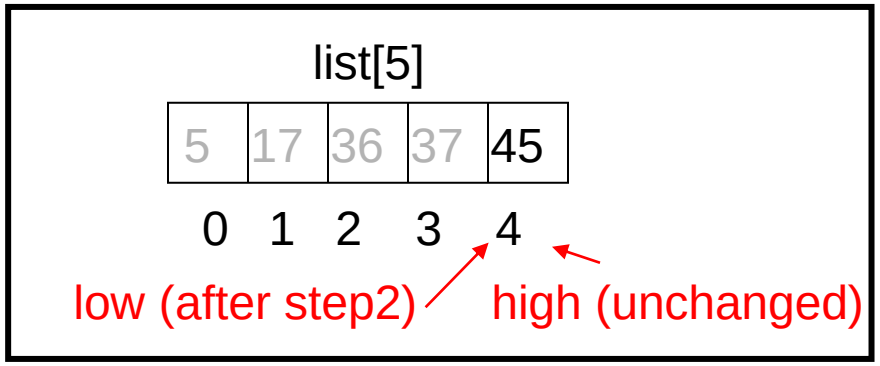
```
int list[5]={5,17,36,37,45};
```

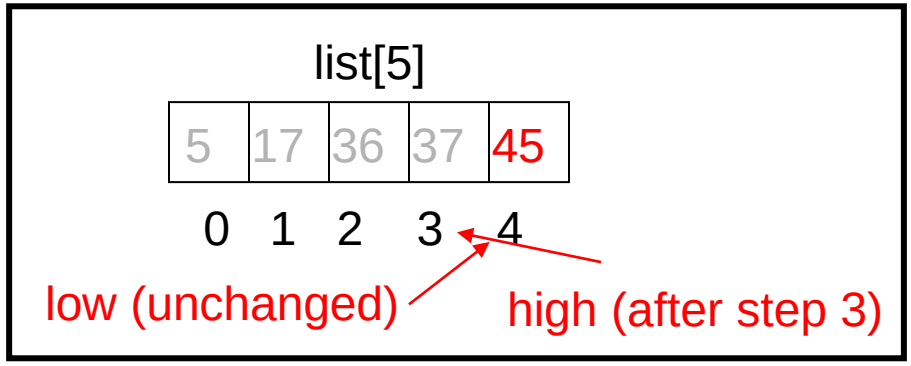
low=0, high=4 key=44

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 and return -1 (not found)

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





Binary Search

Example

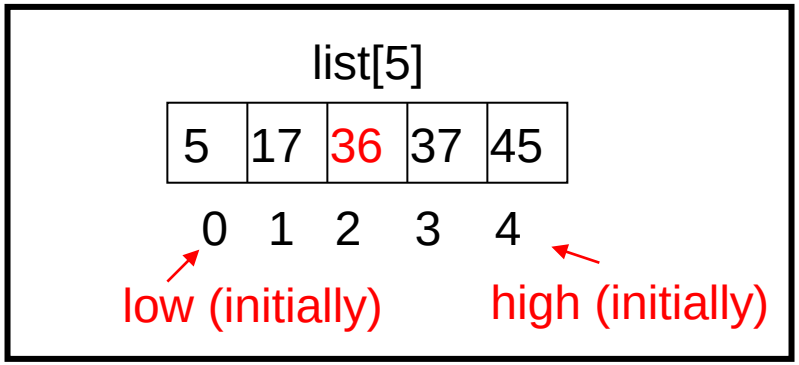
```
int list[5]={5,17,36,37,45};
```

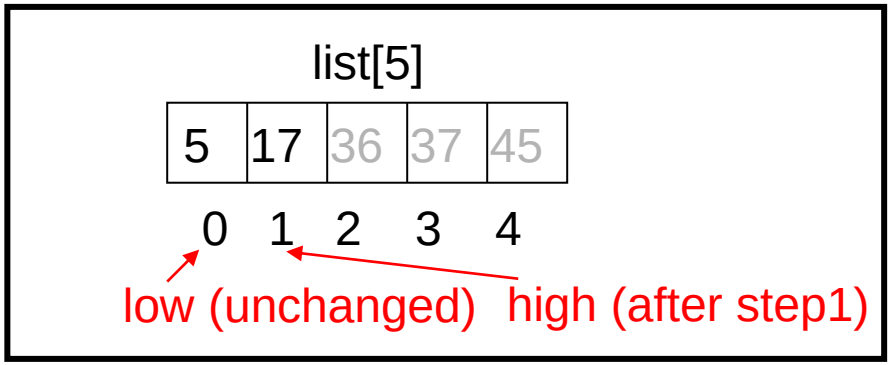
low=0, high=4 key=5

(the same example with different key)

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

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





Binary Search

Example

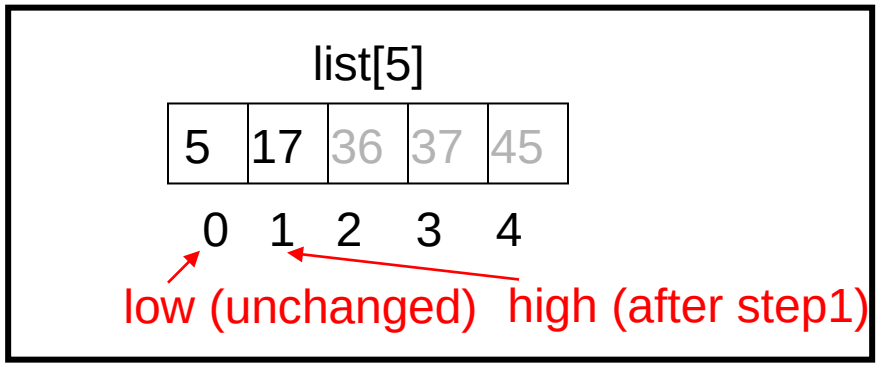
```
int list[5]={5,17,36,37,45};
```

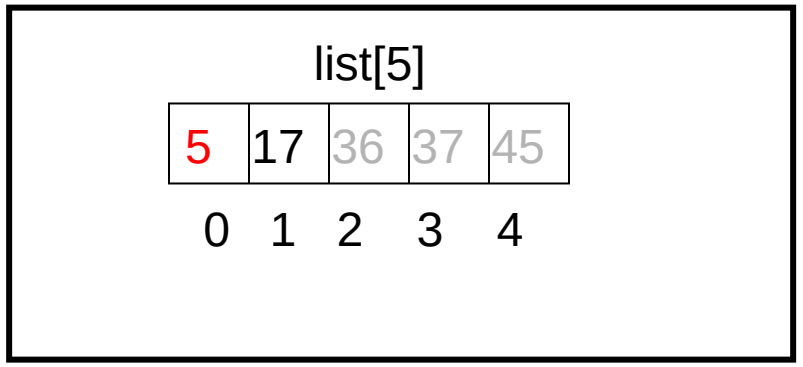
low=0, high=4 key=5

(the same example with different key)

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

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





Binary Search

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

(ceiling) 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:

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

$$1\text{msec} = k \cdot \lceil \log_2 100 \rceil$$

$k = 1/7$ msec/comparison

Hence, $t = \left(\frac{1}{7}\right) \cdot \lceil \log_2 n \rceil$

$$t_{500} = \left(\frac{1}{7}\right) \cdot \lceil \log_2 500 \rceil = 9/7 \cong 1.29\text{msec}$$

$$t_{20000} = \left(\frac{1}{7}\right) \cdot \lceil \log_2 20000 \rceil = 15/7 \cong 2.1\text{msec}$$

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

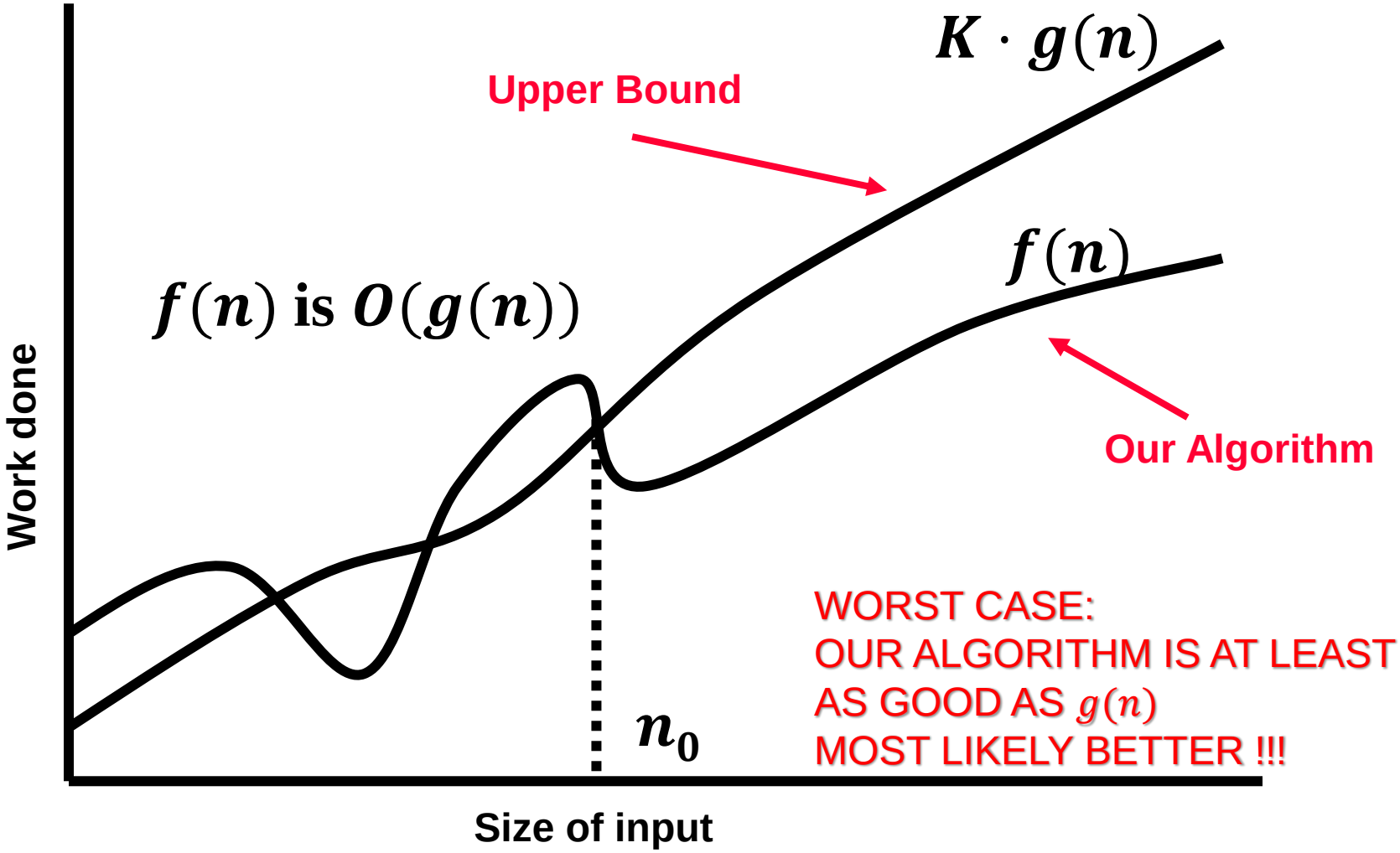
Assume that $g(n)$ is a strictly positive function for large enough n. Function $f(n)$ is $O(g(n))$ if there exists a constant K and some n_0 such that

$$0 \leq \mathrm{f(n)} \leq K \cdot g(n) \text{ for all } n \geq n_0$$

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 $\log_a n = k \cdot \log_b n$ for any $a, b \in \Re$)
  - n, n^k (polynomial)
  - k^n (exponential)

Big-O Notation

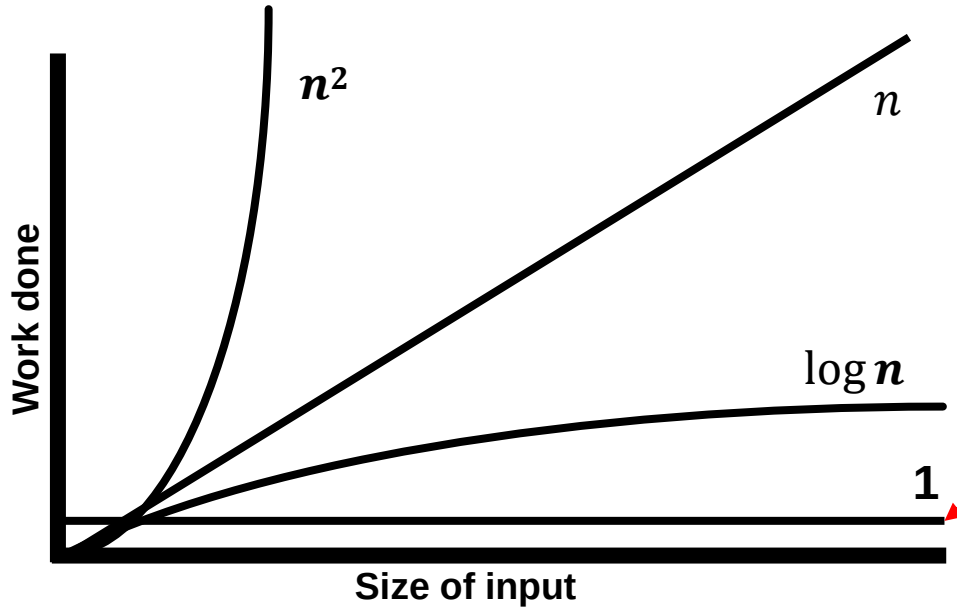


Comparing Two Algorithms

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 is 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

- Example 1
  - $f(n) = n^2 + 250n + 106$ is $O(n^2)$

    because

    $f(n) \leq n^2 + n^2 + n^2 = 3 \cdot n^2$ for $n \geq 103$

    K n_0
- Example 2
  - $f(n) = 2^n + 10^{23}n + \sqrt{n}$ is $O(2^n)$

    because

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

    $f(n) \leq 3 \cdot 2^n$ for $n > n_0$

    K

No Uniqueness

- There is no unique set of values for n_0 and K in proving the asymptotic bounds
- Example: Prove that $100n + 5 = O(n^2)$
- Computation 1
  - $100n + 5 \leq 100n + n = 101n \leq 101n^2$

    for all n ≥ 5

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

    for all n ≥ 1

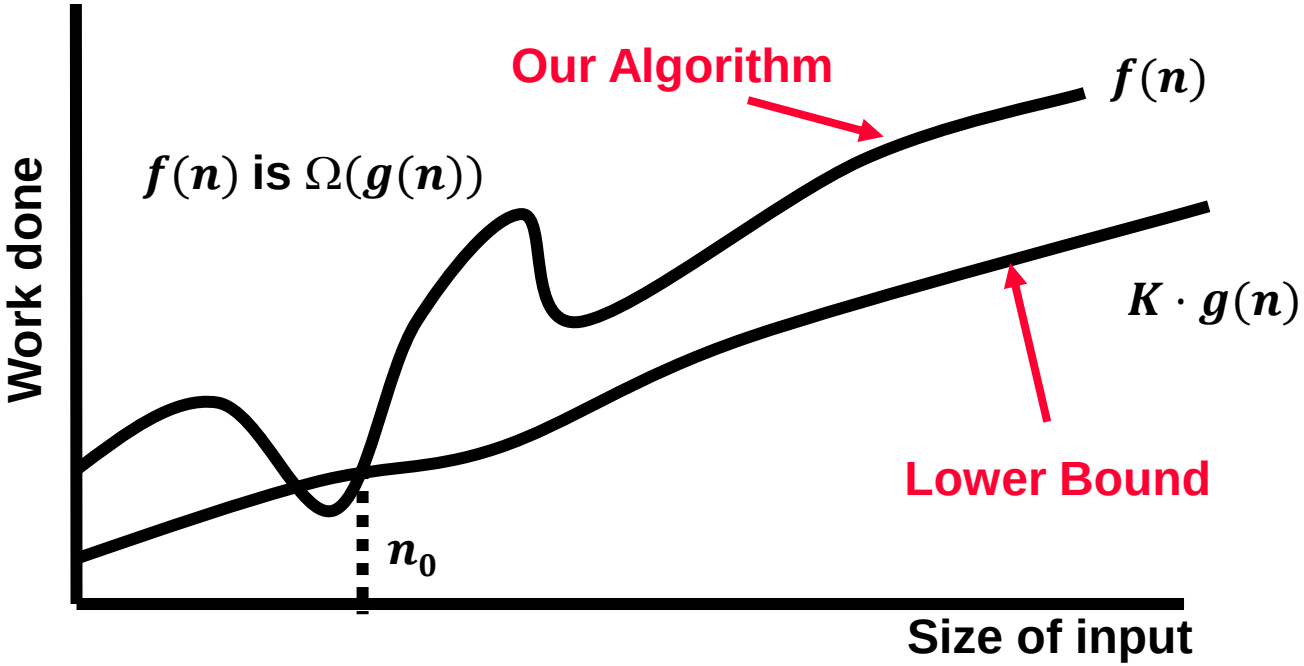
    $\rightarrow n_0 = 1$ and $K = 105$ is also a solution
- Must find SOME constants K and n_0 that satisfy the asymptotic notation relation

Big-Omega Notation

Assume that $g(n)$ is a strictly positive function for large enough n. Function $f(n)$ is $\Omega(g(n))$ if there exists a constant K and some n_0 such that

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

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



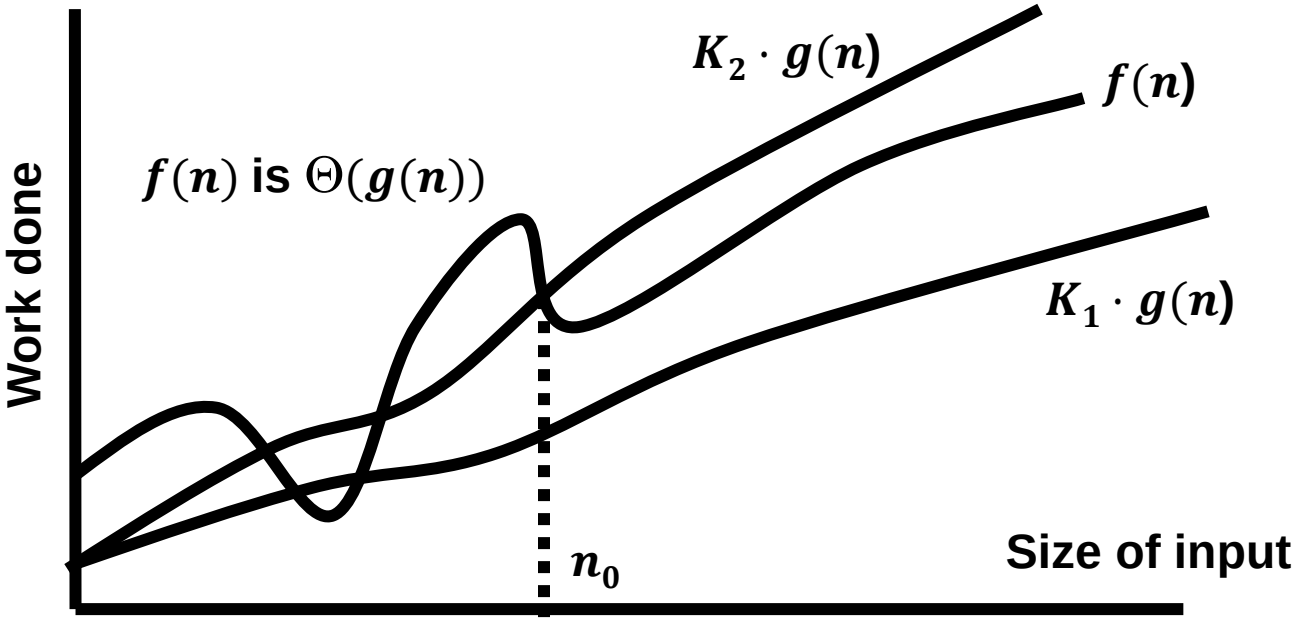
BEST CASE:
OUR ALGORITHM IS AT MOST AS GOOD AS $g(n)$
MOST LIKELY WORSE!!!

Big-Theta Notation

Assume that $g(n)$ is a strictly positive function for large enough n. Function $f(n)$ is $\Theta(g(n))$ if there exist constants K_1 and K_2 and some n_0 such that

$$K_1 \cdot g(n) \leq f(n) \leq K_2 \cdot g(n) \text{ for all } n \geq n_0$$

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 $\Omega(g(n))$ implies: $f(n)$ "≥" $g(n)$
- Θ notation: asymptotic "equality": TIGHT BOUND
  - $f(n)$ is $\Theta(g(n))$ implies: $f(n)$ "=" $g(n)$
- Theorem

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

$\rightarrow f(n)$ 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
- Examples
  - Assume $f(n) = \log(n),\ g(n) = n^2\ , h(n) = n!$
  - $f(n)$ is $O(g(n))$
  - $g(n)$ is $O\big(h(n)\big)$

  $\Rightarrow f(n)$ is $O(h(n)) = O(n!)$ since $f(n)$ is $O(g(n))$ and $g(n)$ is $O\big(h(n)\big)$

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: $a_0 + a_1 \cdot n + \ldots + a_d \cdot n^d$ 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
- Logarithms: $O(\log_a n) = O(\log_b n)$ for any constants $a, b > 0$

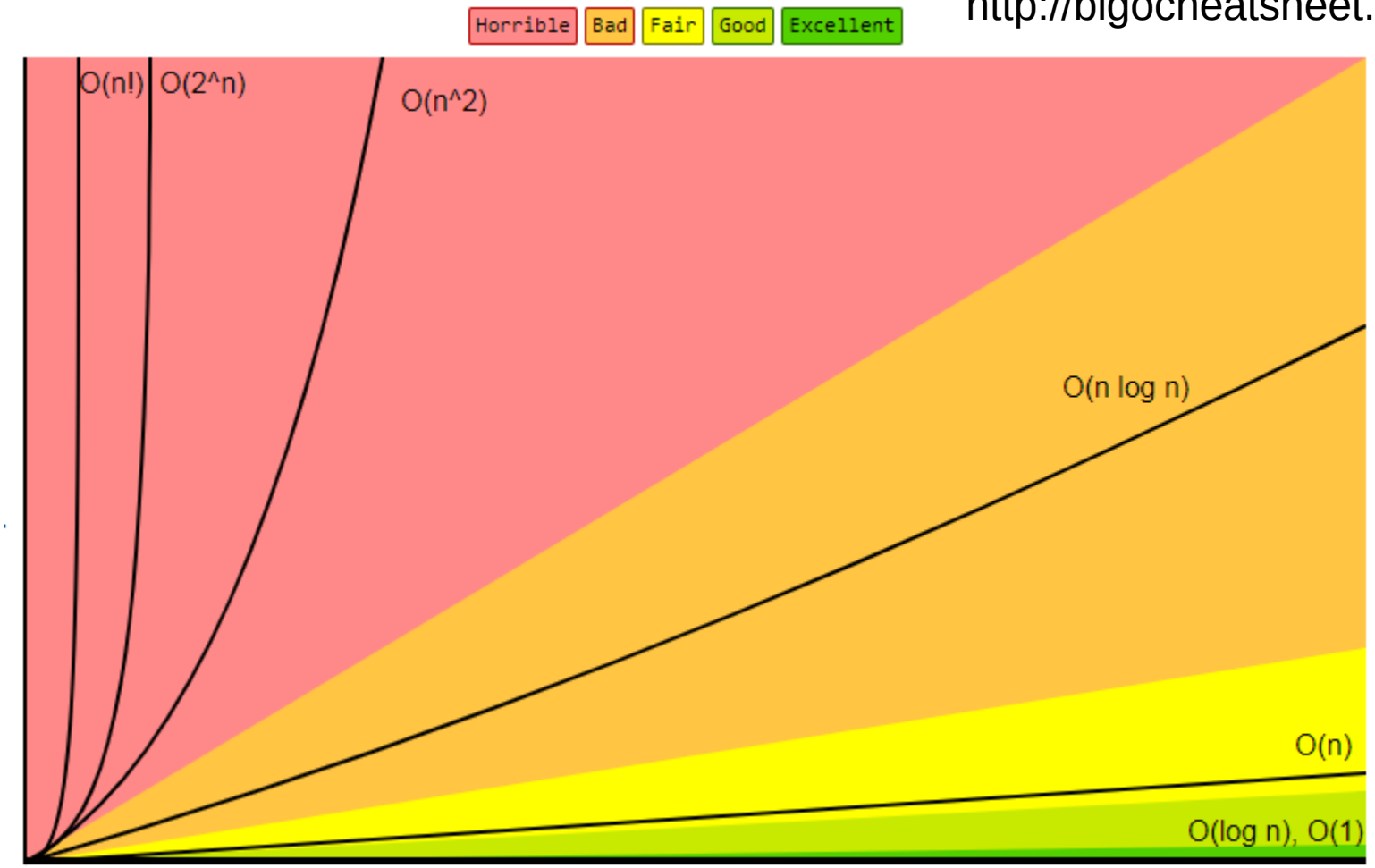
  $\rightarrow$ So, you can state logarithms without base!
- Fact: For every $\mathrm{d} > 0$, $\log n = O(n^d)$

  $\rightarrow$ Every polynomial grows faster than every log
- Exponentials: For every $r > 1$ and every $d > 0$, $n^d = O(r^n)$

  $\rightarrow$ Every exponential grows faster than every polynomial

Compare

http://bigocheatsheet.com/



Compare

http://bigocheatsheet.com/

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Example

- $f(n) = \frac{1}{2}n^2 + 3 \cdot n$ is $\Theta(n^2)$
- To show this, 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 expressions by n^2
- $K_1 \leq \frac{1}{2} + 3\frac{1}{n} \leq K_2$

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

Example

```
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;
    }
}
```

- Assume
  - myfunc1: $\Theta(n)$
  - myfunc2: $\Theta(n^2)$
  - myfunc3: $O(1)$
- If x is always even: n times execute myfunc1+myfunc3
  $n\big(\Theta(n)+O(1)\big)=\big(\Theta(n^2)+\Theta(n)\big)=\Theta(n^2)$
- If x is always odd: n times execute myfunc2+myfunc1
  $n\big(\Theta(n^2)+\Theta(n)\big)=\Theta(n^3)+\Theta(n^2)=\Theta(n^3)$
- Worst case
  - x is always odd: $\Theta(n^3)\Rightarrow O(n^3)$
- Best case
  - x is always even: $\Theta(n^2)\Rightarrow\Omega(n^2)$

    $\rightarrow$ 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
        int 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
}
```

$$T(n) = \mathrm{t1} + \mathrm{t2} + \mathrm{t3a} + (n+1)\cdot \mathrm{t3b} + n\cdot(\mathrm{t3c} + \mathrm{t4} + \mathrm{t5}) + \mathrm{t6}$$

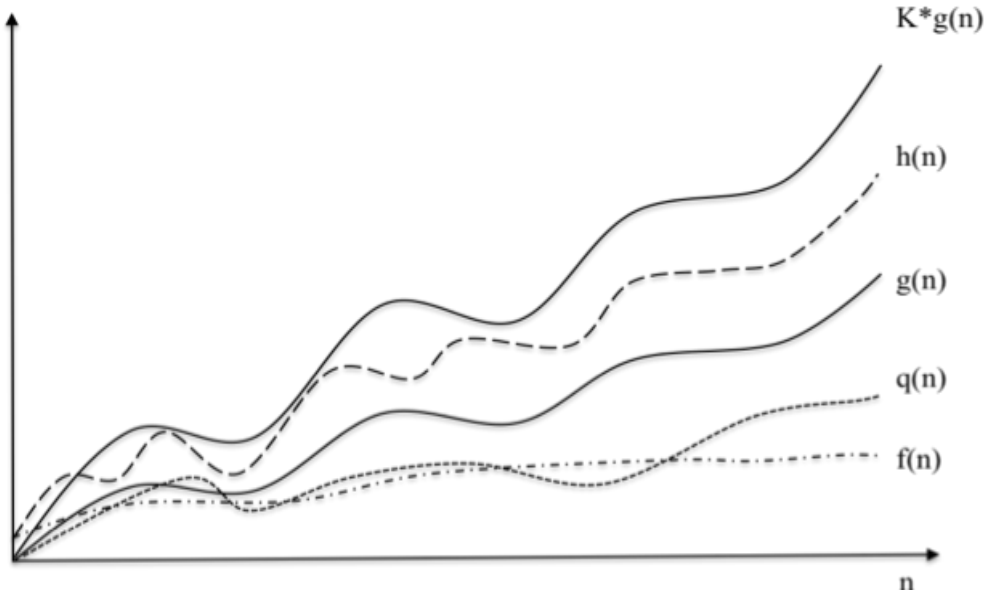
$$T(n) = \mathrm{T_A} + n\cdot \mathrm{T_B}$$

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

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:

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

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

$$p(n): O(p(n)) = O(h(n) + f(n)) = O(g(n))$$
$$\Omega(p(n)) = \Omega(h(n) + f(n)) = \Omega(g(n))$$
$$\Big\} \Theta(g(n))$$

Practical Use: Scheduling at Switches

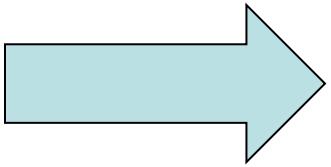
Algorithm (Year)	Complexity	Fairness (WFI)
GPS (1993)	NA	0 (Perfect)
DRR (1996)	O(1)	O(N)
WFQ (1993)	O(N)	O(N)
SCFQ (1994)	O(logN)	---
WF^2Q (1996)	O(N)	O(Li/ri)
WF^2Q+ (1997)	O(logN)	O(Li/ri)

GPS: generalized processor sharing
DRR deficit round robin
WFQ weighted fair queuing
SCFQ self clocked fair queuing
Li: longest packet size
ri: allocated rate

Practical Use: Allocation of Virtual Machines to Servers

Virtual Machine (VM) Requests

Problem: Allocating M VM requests on N Physical servers in a Cloud



Cloud Server Infrastructure

N. U. Ekici, K. W. Schmidt, A. Yazar and E. G. Schmidt, "Resource Allocation for Minimized Power Consumption in Hardware Accelerated Clouds,"International Conference on Computer Communication and Networks (ICCCN), 2019, pp. 1-8.

$O(N^M)$		Number of PMs (N)								
		10	20	50	100	150	200	250	300	400
Number of Requests (M)	2	0.01	0.01	0.02	0.03	0.03	0.04	0.05	0.06	0.07
	5	0.02	0.03	0.05	0.09	0.13	0.30	0.35	0.45	0.65
	7	0.03	0.05	0.12	0.42	0.71	1.00	1.28	1.88	2.95
	10	0.16	0.60	1.33	4.01	6.70	10.93	18.84	28.79	46.81
	15	0.43	1.83	4.49	29.69	35.04	37.99	40.08	48.48	49.98
	20	4.69	14.18	70.85	141.10	91.25	182.67	515.91	646.02	181.88
	25	113.24	157.12	1990.15	NA	NA	NA	NA	NA	NA

Time for computing the decision in seconds.
Results shown in black are average of 10 runs.
Red results are from a single run.
Solver: TOMLAB toolbox for MATLAB, CPLEX solver.

Complexity of Recursive Functions: Factorial

T(n)

```
int Factorial (int n){
        if (n==0)//stopping condition is 0!
                return 1;
        else //recursive step
                return n*Factorial(n-1);
}
```

tB

$T(n-1) + tA$: There is some additional overhead on top of calling Factorial $(n-1)$

- $T(0) = tB$: stopping condition
- $T(n) = tA + T(n-1)$: recursive step
- $T(n) = T(n-1) + tA = T(n-2) + 2tA = \cdots = T(0) + ntA = tB + ntA$
  $\rightarrow O(n)$
- tA and tB are $O(1) = \theta(1)$ DOES NOT DEPEND ON n

Recursion Example: Fibonacci Numbers

- Fibonacci numbers
  - $F_n = 0, \text{if } n = 0$
  - $F_n = 1, \text{if } n = 1$
  - $F_n = F_{n-1} + F_{n-2}, \text{if } n \geq 2$
- Non-recursive (iterative) program

```
int Fibonacci(int n){
  int sum;
  int prev=-1;
  int result=1;
  for(int i=0;i<=n;i++){
    sum=result+prev;
    prev=result;
    result=sum;
  }
  return result;
}
```

- Recursive program

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

Complexity of Recursive Functions: Fibonacci Numbers

Non-recursive program:

```
int Fibonacci(int n){
  int sum;
  int prev=-1;
  int result=1;
  for(int i=0;i<=n;i++){
    sum=result+prev;
    prev=result;
    result=sum;
  }
  return result;
}
```

for-loop with *n* iterations and constant operations

→ O(n), Ω(n) hence θ(n)

Recursive program:

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

Additional overhead

$$T(n) = \theta(1)\ if\ n < 2$$
$$T(n) = T(n-1) + T(n-2) + \theta(1), n \geq 2$$
$$T(1) = \theta(1)$$
$$T(2) = \theta(1)$$
$$T(3) = T(2) + T(1) + \theta(1)$$
$$T(3) = \theta(1) \cdot (1 + 1 + 1) = 3 * \theta(1)$$
$$T(4) = T(3) + T(2) + \theta(1)$$
$$T(4) = 3\theta(1) + \theta(1) + \theta(1) = 5 * \theta(1)$$
$$T(n) = \theta(1) \cdot Fib(n) = \theta(Fib(n))$$

Complexity of Recursive Functions: Fibonacci Numbers

- Another solution: Upper Bound

$$T(n) = \theta(1)\ if\ n < 2$$
$$T(n) = T(n-1) + T(n-2) + \Theta(1), n \geq 2$$
$$T(n) = T(n-1) + T(n-2) + \Theta(1)$$
$$\leq 2T(n-1) + \Theta(1) \qquad [T(n-2) \leq T(n-1)]$$
$$\leq 2(2T(n-2)) + 3\Theta(1) = 2^2 T(n-2) + (2^2 - 1)\Theta(1)$$
$$\ldots$$
$$\leq 2^{n-2}T(2) + (2^{n-2} - 1)\Theta(1)$$
$$\leq 2^{n-1}T(1) + (2^{n-1} - 1)\Theta(1)$$
$$= 2^{n-1}\Theta(1) + (2^{n-1} - 1)\Theta(1)$$
$$\Rightarrow T(n) \text{ is } O(2^n)$$

Complexity of Recursive Functions: Fibonacci Numbers

- Lower Bound
  - Case: n is even

$$T(n) = T(n-1) + T(n-2) + \Theta(1) \geq 2\big(T(n-2)\big)$$

$$[T(n-1) \geq T(n-2)]$$

$$\geq 2^2 T(n-4) \cdots \geq 2^{\frac{n-2}{2}} T(2) = \Omega(2^n)$$

  - Case: n is odd

$$T(n) = T(n-1) + T(n-2) + \theta(1) \geq 2\big(T(n-2)\big) \geq 2^2 T(n-4)$$

$$\cdots \geq 2^{\frac{n-1}{2}} T(1) = \Omega\,(2^n)$$

- Overall: $T(n)$ is $\Omega(2^n)$ and $O(2^n) \Leftrightarrow T(n)$ is $\Theta(2^n)$

  i.e. Exponential, so infeasible

Space Complexity

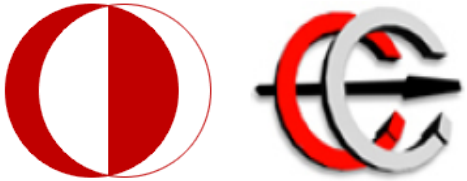
- Definition of Space Complexity
  - Total memory required by an algorithm, as a function of input size
    - Memory for variables and constants
    - Memory for data structures (arrays, hash maps, DP tables)
    - Memory for function calls / recursion stack
    - Memory for auxiliary buffers / temporary storage
- Importance of Space Complexity
  - Determines whether an algorithm can fit in limited memory
  - Space can be traded with time (e.g., recomputing vs storing results)
  - Some optimizations reduce space without affecting correctness

Complexity of Fibonacci with Memoization

- Time Complexity
  - Without memoization: $O(2^n)$
  - With memorization: O(n)
    - Each Fibonacci number from 0 to n is computed at most once.
    - Each computation takes O(1) work (just a couple of additions/lookups)
- Space Complexity
  - Memoization array: O(n) to store results
  - Recursion stack: in the worst case depth = O(n)
  - Overall: O(n) space complexity

Complexity of Knapsack with Tabulation

- Time Complexity
  - Without tabulation: $O(2^n)$
  - With memorization: O(n·W)
    - There are n items and W possible capacities
    - For each pair (i, w) we do constant-time work (max, +, lookup)
- Space Complexity
  - Standard DP table = (n+1) · (W+1) entries: O(n·W)
  - With row-optimization (1D DP using only the previous row): O(W)



EE 441 Data Structures

Lecture 4: Algorithm Complexity