EE 441 Data Structures Lecture 6

Algorithm complexity

Problem and Algorithm

- An algorithm is a step-by-step procedure.
- A problem is the thing an algorithm "solves".
- A problem consists of
  - A domain containing instances of the problem
  - A question that can be asked about any of the instances
- An algorithm solves problem P if, given an instance I of P as input, it generates the answer for P's question for I.
- For some (not all) problems, an algorithm can be developed that solves every instance of P.

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???*
- Predicting the resources that the algorithm requires
  - Resources: memory, communication bandwidth, hardware but MOSTLY TIME
- You can run the algorithm and see the efficiency!!
- Benchmarking:
  - Run the program and measure runtime
    - 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 input, is it possible to determine how an algorithm’s runtime changes?

Analysis of an Algorithm

- In general the run time of a given algorithm grows by the size of the input
- Growth rate: How quickly the time of an algorithm grows as a function of the problem 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.
  - Pathological instances determine complexity even though they may be very rare.
  - Guarantees an upper bound, but not very useful for a particular instance unless the bound is tight.
- Best case
  - Provides a lower bound on running time.
  - Input is the one for which the algorithm runs the fastest.

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

Average Case Analysis

- 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.

- Find worst cases.
  - Some algorithms perform well for most cases but are very inefficient for few inputs.

- An algorithm is said to be “good” if its worst-case time complexity is bounded by a poynomial function of *n.*

- For simplicity, any algorithm that is not polynomially bounded is referred to as exponential.

- Some algorithms are well-behaved, i.e. predictable in number of steps required for a problem instance of size *n.*

- Other algorithms may have varying number of steps for problem instances of the same size.

- For such problems we can either count the average or the maximum steps for instances of size *n.*

Measuring Efficiency

- 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.
- Number of steps required to solve an instance, maximized over all instances of size *n*, and expressed as a function of *n.*

Example

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

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

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

→t1

→t2a t2b t2c

→t3

→t3

Time it takes to run:

T(n)=t1+t2a+(n+1)t2b+nt2c+t3+t4

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

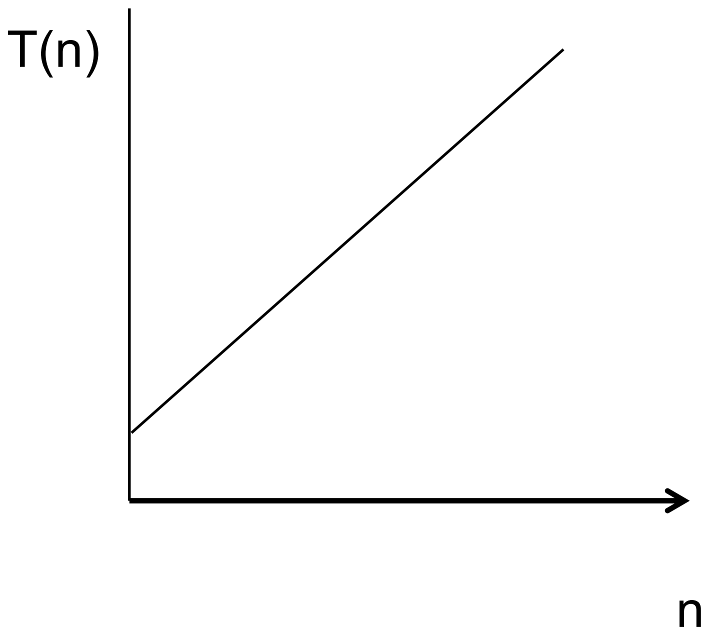
Rate of Growth

- Remember:

$T(n)=t1+t2a+(n+1)t2b+nt2c+t3+t4$

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

$T(n)=T_A n+T_B$



As n goes to infinity:

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!!

Example Algorithm to solve a problem

- Problem:
  - An 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)

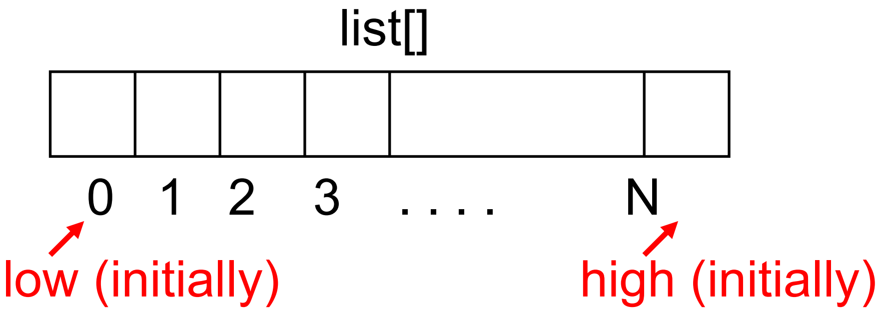
- Search for 37
- Middle is 36
- If 37 exists it has to be in the higher part of the array

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

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

list[5]

5	17	36	37	45
0	1	2	3	4

low (initially) high (initially)

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

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

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

list[5]

5	17	36	37	45
0	1	2	3	4

low (initially) high (initially)

e.g.
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

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

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

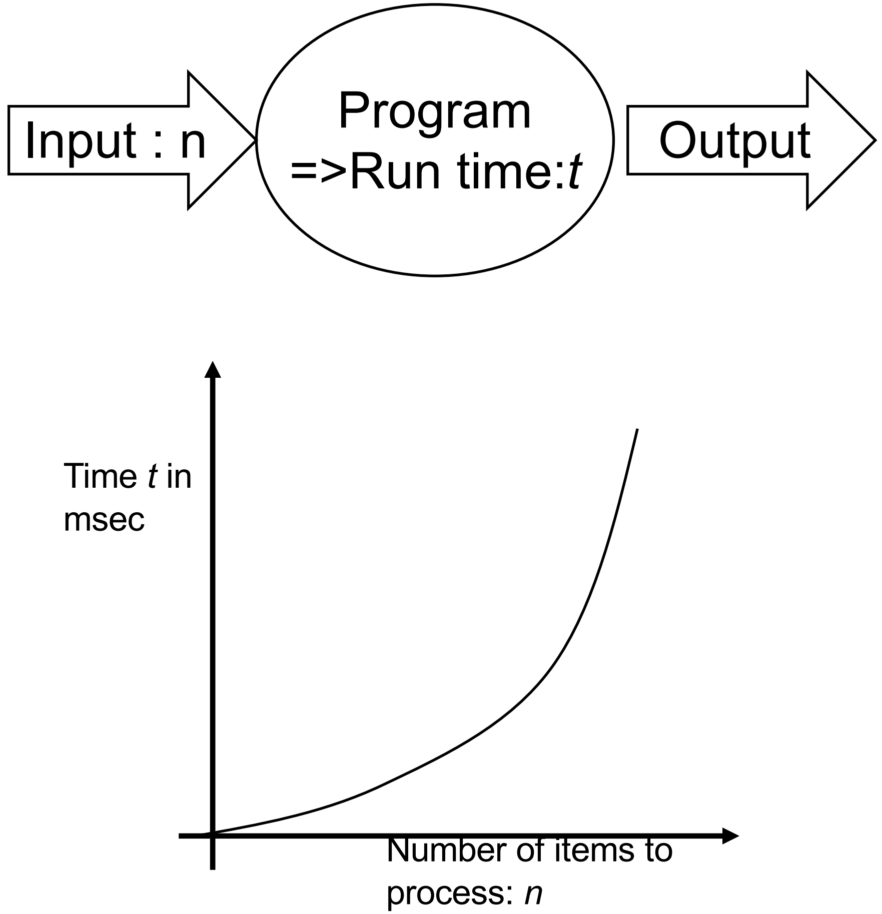
$t_{500}=(1/7)*\lceil \log_2 500 \rceil=9/7\cong1.29msec$

$t_{20000}=(1/7)*\lceil \log_2 20000 \rceil=15/7\cong2.1msec$

Run time vs problem size

N	sequential search O(n)	binary search O(logn)
2	2	1
8	8	3
16	16	4
64	64	6
100	100	7
128	128	7
1000	1000	10
1024	1024	10
64000	64000	16
65536	65536	16

Algorithm Complexity



Computational Complexity Metrics

- 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

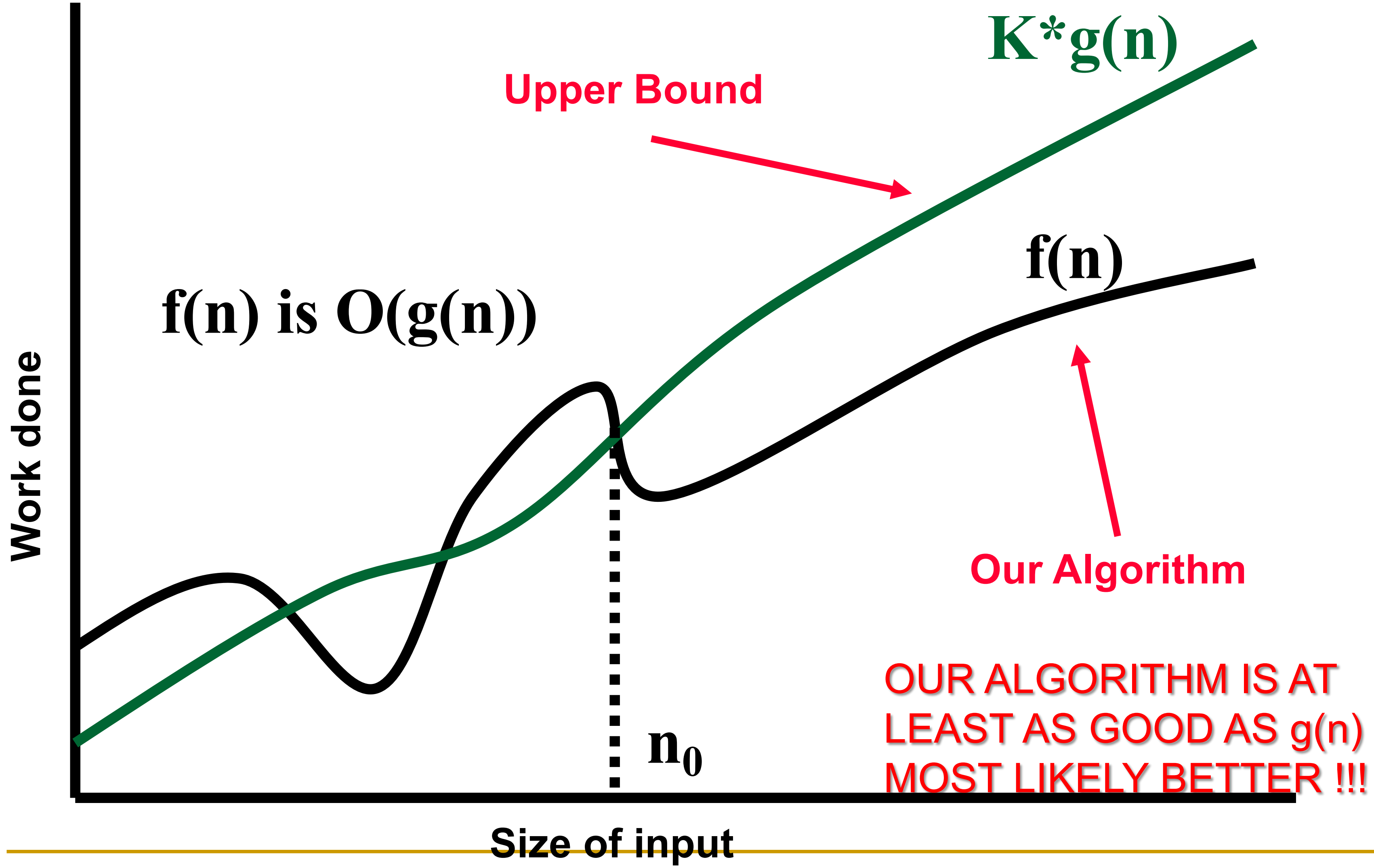
Function $f(n)$ is $O(g(n))$ if there exists a constant K and some n_0 such that

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

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

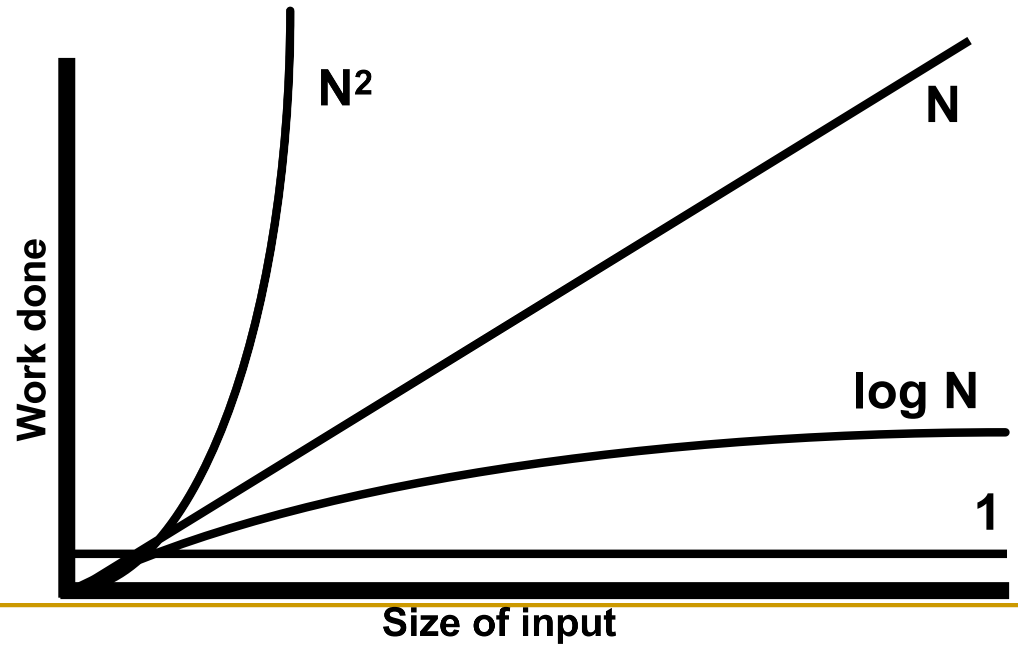
- Usually, g(n) is selected among:
  - log n (note $\log_a n = k * \log_b n$ for any $a, b \in \Re$)
  - n, n^k (polynomial)
  - k^n (exponential)

Big-O Notation



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



Examples

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

because

$f(n) \le n^2+n^2+n^2$ 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

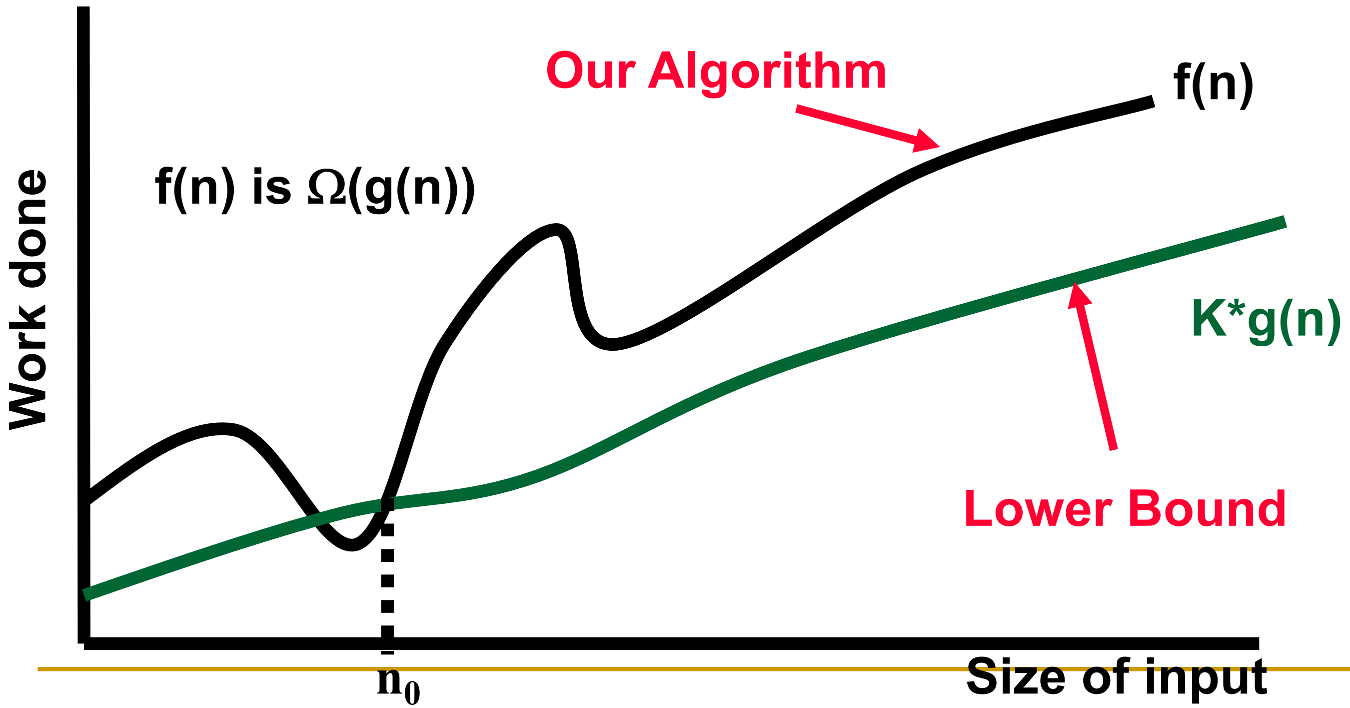
There is no unique set of values for n_0 and K in proving the asymptotic bounds

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



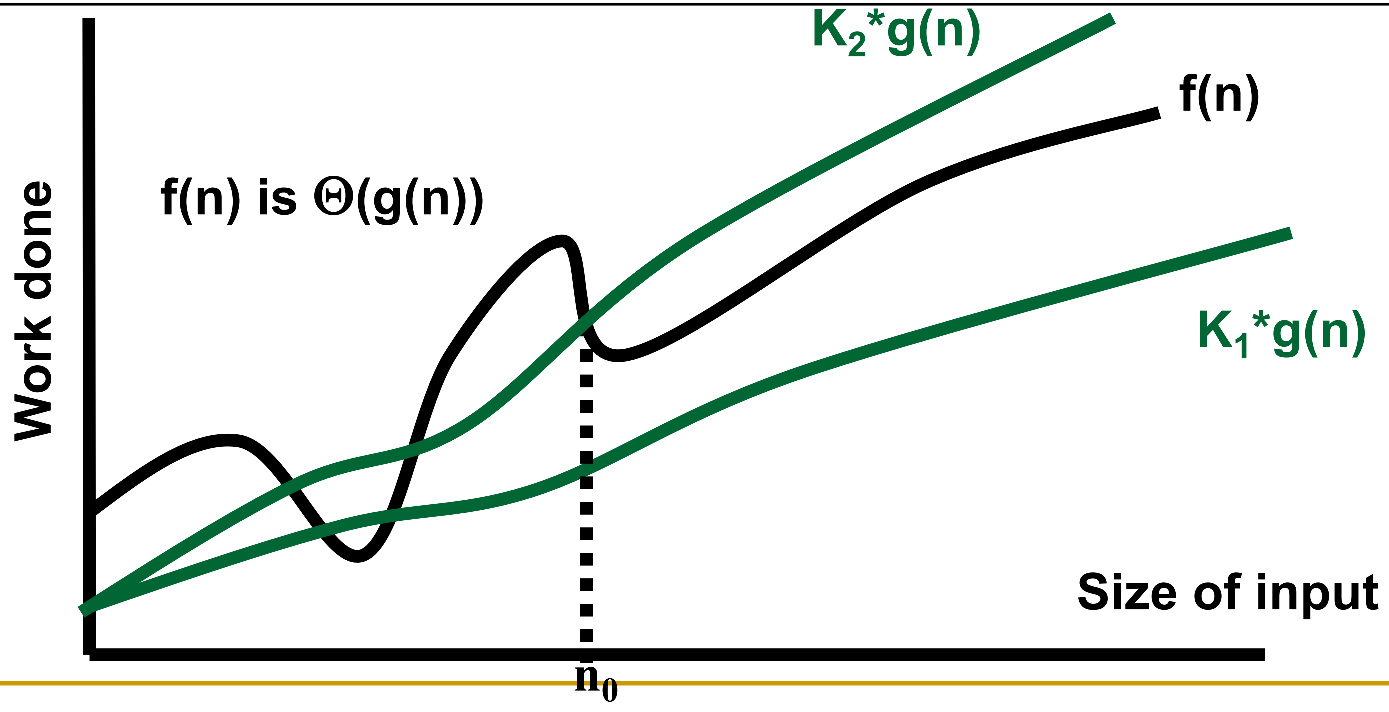
OUR ALGORITHM IS AT MOST AS GOOD AS g(n)
MOST LIKELY WORSE!!!

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

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

i.e., as n→∞, f(n) is upper and lower bounded by some constants times g(n).



Asymptotic Notation

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

Properties

- *Theorem:*

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

f(n) is Θ(g(n)) if f(n) is both O(g(n)) and Ω(g(n))

- **Transitivity:**
  - $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$
  - Same for O and Ω
- 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

Properties

- 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. $a_0 + a_1 n + \ldots + a_d n^d$ is $\Theta(n^d)$ if a_d is not 0.
- Polynomial time. Running time is $O(n^d)$ for some constant d independent of the input size n.
- 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, $\log n <= O(n^x)$.
  - log grows slower than every polynomial

Common Asymptotic Bounds

- Exponentials. For every $r > 1$ and every $d > 0$, $n^d >= O(r^n)$.
  - every exponential grows faster than every polynomial
  - There always is an *N* such that for all $n \geq N$, a polynomial-time algorithm is better than an exponential one

- An algorithm takes T(n) μs to compute n input

T(n)	n=16	n=256
$\log_2 n$	4μs	8μs
n	16μs	256μs
n^2	256μs	65.5ms
2^n	65.5ms	10^{63} years

Advantages of Polynomial time algorithms

- Polynomial-time algorithms take better advantage of advances in computer technology, e.g.
  - Suppose two algorithms with complexities $O(n^3)$ and $O(2^n)$ can solve a problem of size n = 100 in one hour.
  - With a computer twice as fast, polynomial-time algorithm can solve instances of size n = 126, whereas exponential algorithm can solve only n = 101.
- Polynomials have nice mathematical properties, e.g. addition and multiplication of two polynomials are still polynomials

- An $O(1.001^n)$ exponential algorithm is better than $O(n^{100})$ or $O(10100n)$ for practical purposes

- Finding the first polynomial-time algorithm for an exponential problem is a big jump, since then it can be improved to find a version of practical use

Algorithm Complexity Examples

Example: $\sum_{i=1}^{n}\sum_{j=1}^{i} i*j$

```
int DSum (int n)
{
    int result = 0;
    for (int i = 1; i <= n; ++i)
        for (int j=1;j<=i,++j)
            result += i*j;
    return result;
}
```

	int DSum (int n)	
	{	
1	int result = 0;	t1
2	for (int i = 1; i <= n; ++i)	t2a, t2b,t2c
3	for (int j=1;j<=i,++j)	t3a, t3b, t3c
4	result += i*j;	t4
5	return result;	t5
	}	

$t1 + t2a + (n+1)\ t2b + n * t2c +$

$$\sum_{i=1}^{n} (t3a + (i+1) * t3b + i * t3c + i * t4)$$

$+t5$

$= t1 + t2a + t2b + t5 + n*(t2b+t2c+t3a+t3b) + (n*(n+1)/2)*(t3b+t3c+t4)$

$= tA + n*tB + n^2*tC \Rightarrow O(n^2),\ \Omega(n^2),\ \theta(n^2)$

- Example:

```
for (i=0;i<N;i++)
   for (j=0;j<i;j++)
     for (k=0;k<5;k++)
            statement x;
//takes t msec to execute statement x
```

Total time:

$$5t\sum_{i=1}^{N} i = 5t\left[N\frac{(N+1)}{2}\right]$$

$O(n^2)$, $\Omega(n^2)$, $\theta(n^2)$

Example:

- myfunc1: $\theta(n)$
- myfunc2: $\theta(n^2)$

```
int Randomfunction (int n, int seed)
{
int x=Rand(seed);
for(int i=1;i<=n;i++)
   if(x%2==0)
     x=Rand(x)+myfunc1(x,n);//  θ(1)+θ(n)=θ(n)
   else
     x=Rand(x)+myfunc2(x,n);//  θ(1)+θ(n²)=θ(n²)
return x;
}
```

repeated n times

- Upperbound: $O(1)+n*\max(O(n), O(n^2)) = O(n^3)$
- Lowerbound: $\Omega(1)+n*\min(\Omega(n), \Omega(n^2)) = \Omega(n^2)$

Complexity of recursive functions

- Example

```
int Factorial (int n)
{
if (n==0)
   return 1;
else
   return n*Factorial(n-1);
}
```

T(n)

tB

T(n-1)+tA

- T(n) = tB: stopping condition
- T(n) = tA+T(n-1): recursive step
- T(n)=T(n-1)+tA=T(n-2)+2tA=…=T(0)+ntA=
- =tB + ntA→O(n)

Complexity of recursive functions

- Example

```
int Power (int n, int x)
{
if (n==0)
   return 1;
else if(n%2==0)//n is even
   retun Power(x*x, n/2);
else//n is odd
   retun x*Power(x*x, n/2);
}
```

- T(n) = ta: stopping condition
- T(n) = tb+T(└n/2┘):n even
- T(n) = tc+T(└n/2┘):n odd
- tb<tc

Complexity of recursive functions

- Suppose $n=2^k$, $k=\log_2 n$:
  - $T(2^k)=tb+T(2^{k-1}) = 2tb+T(2^{k-2}) = \ldots = k*tb+T(2^0) = k*tb+tc+T(0) = k*tb+tc+ta$
  - $T(n) = \log_2 n*tb+tc+ta$

- Suppose $n=2^k-1$, $k=\log_2(n+1)$:
  - $T(2^k-1) = tc+T(2^{k-1}) = tc+tb+T(2^{k-2}) = \ldots = tc+(k-1)*tb+T(2^0)$

$= tc+(k-1)*tb+ta$

  - $T(n) = \log_2(n+1)*tb+ta+tc$

- $\theta(\log_2 n)$

Fibonacci

- Example: Fibonacci numbers
  - Fn=0, n=0
  - Fn=1, n=1
  - Fn-1+Fn-2, n≥2

- Complexities:

O(n), Ω(n) hence θ(n)

```
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 Fibonacci

- Recursive program:

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

- $T(n) = \begin{cases} \theta(1) & n < 2 \\ T(n-1)+T(n-2)+\theta(1) & n \geq 2 \end{cases}$

- Notice that $T(n) = T(n-1)+T(n-2) + \theta(1) \leq 2T(n-1) \leq 2^2 T(n-2)+2\theta(1)$ ....

$\leq 2^{n-2}T(2) +(n-2)\theta(1) = 2^{n-2}\theta(1)+(n-2)\theta(1)$

- $T(n)$ is $O(2^n)$

This time it is better to indicate a lower bound rather than an upper bound, i.e. Ω(.)

- Case: n is even

$T(n)= T(n-1)+T(n-2) +\theta(1) \geq 2(T(n-2)) \geq 2^2\ T(n-4)\ldots \geq 2^{(n-2)/2}T(2) \Rightarrow \Omega(2^{n/2})$

- Case: n is odd

$T(n)= T(n-1)+T(n-2) +\theta(1) \geq 2(T(n-2)) \geq 2^2\ T(n-4)... \geq 2^{(n-1)/2}T(1) \Rightarrow \Omega(2^{n/2})$

- So T(n) is $\Omega(2^{n/2})$ i.e. Exponential, so infeasible

- In fact Fib(n) is $\Omega((3/2)^n)$