EE 441 – CH3

COMPLEXITY OF ALGORITHMS

Instructor: UĞUR HALICI

Search an n-element array for a match with a given key; return the array index of the matching element or -1 if no match is found

```
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;
}
```

In the worst case n comparisons are performed. Expected (average) number of comparisons is n/2 and expected computation time is proportional to n.

That is, if the algorithm takes 1 msec with 100 elements, it takes ~5msec with 500 elements, ~200 msec with 20000 elements, etc.

Example: Binary Search Algorithm (using a sorted array)

```
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
}
```

HW:Write a recursive function for binary search

Example (continues)

```
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
}
```

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

Binary Search(list, 0,4,44}

1
mid=(0+4)/2=2
midvalue=36
key>midvalue
low=3
2
mid=(3+4)/2=3
midvalue=37
key>midvalue
low=4
3
mid=(4+4)/2=4
midvalue=45
key<midvalue
high=3
4
since high=3<low=4, exit
return –1 (not found)
```

Example (continues)

```
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
}
```

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


Binary Search(list, 0,4,5}

1
mid=(0+4)/2=2
midvalue=36
key<midvalue
high=1
2
mid=(0+1)/2=0
midvalue=5
key=midvalue
return 0 (found)
```

In the worst case Binary search makes $\lceil \log_2 n \rceil$ comparisons

Example:

n	**8**	**20**	**32**	**100**	**128**	**1,024**	**65,536**
$\lceil \log_2 \rceil$	3	5	5	7	7	10	16

Assume Binary Search 1 msec with 100 elements, then it takes 7 msec with 12,800 elements, 16 msec with 6,553,600 elements

n	**Sequential Search O(n)**	**Binary Search O(log n)**
100	1 msec	1 msec
12,800	128 msec	7 msec
6,553,600	6,553,600 msec	16 msec

Asymptotic upper bound: Big-Oh

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

$f(n) \leq K*g(n)$ for any $n>n_0$. That is as $n \rightarrow \infty$, f(n) is upper bounded by a constant times g(n).

Asymptotic lower bound: Omega

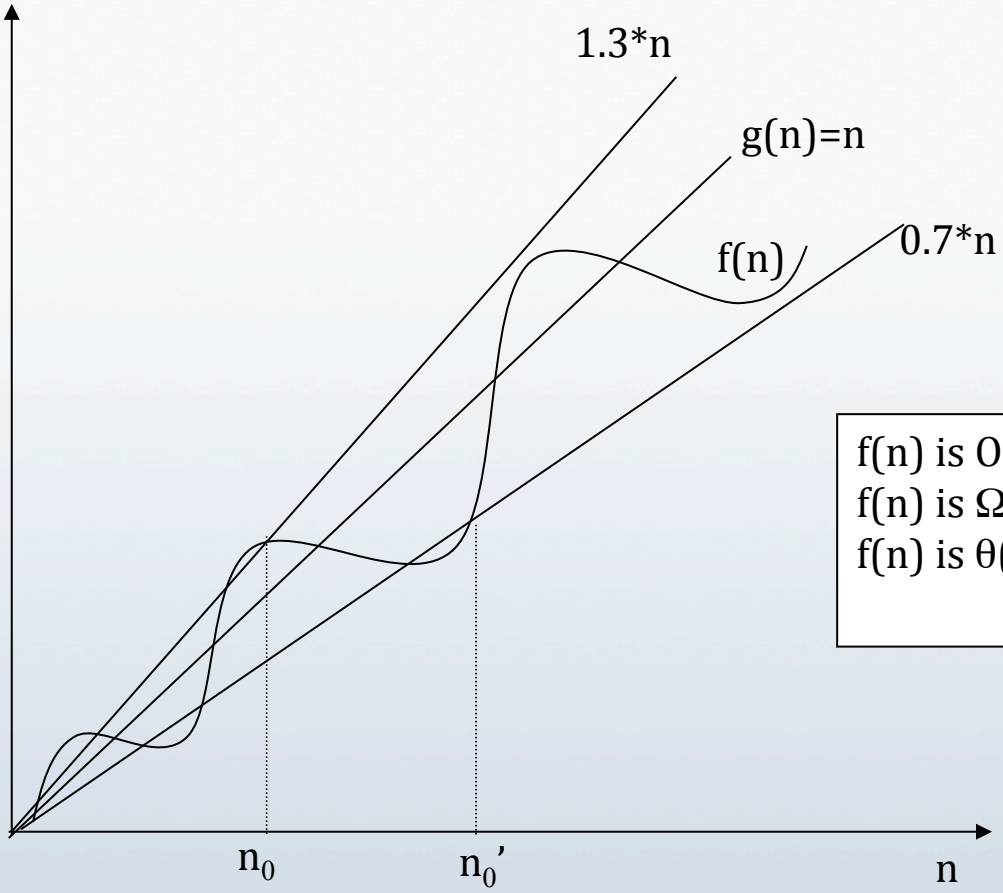
Function f(n) is $\Omega(g(n))$ if there exists a constant K and some n_0 such that

$f(n) \geq K*g(n)$ for any $n>n_0$. That is as $n \rightarrow \infty$, f(n) is lower bounded by a constant times g(n).

Asymptotic upper- lower bound : Theta (Tight bound)

f(n) is $\theta(g(n))$ iff f(n) is O(g(n)) and $\Omega(g(n))$

Example



f(n) is O(n)
f(n) is Ω(n)
f(n) is θ(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
- 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 $g(n) = \Omega(f(n))$

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

Use O(.) to show how good is an algorithm
For example $O(n^2)$, polynomial, feasible

Use Ω(.) to show how bad it is
for example $\Omega(2^n)$, grows exponentially, not feasible

Example: $f(n)= n^3+70n^2+2$ is $O(n^3)$, and also $\Omega(n^3)$, $\theta(n^3)$
Example: $f(n)=2^n+10^{23}n+n^{1000}$ is $\Omega(2^n)$, and also $O(2^n)$, $\theta(2^n)$

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

	int Sum (int n)	
	{	
1	int result = 0;	t1
2	for (int i = 1; i <= n; ++i)	t2a, t2b,t2c
3	result += i;	t3
4	return result;	t4
	}	

f=t1+ t2a + (n+1) t2b + n*t2c +n*t3+t4= tA+n*tB

$\Rightarrow O(n), \Omega(n), \theta(n)$

Example: $\sum_{i=0}^{n} a_i x^i$

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

f=t1+t2+t3a+(n+1)*t3b+n*(t3c+t4+t5)+t6=tA+n*tB $\Rightarrow$ O(n), Ω(n), θ(n)

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

	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
	}	

$$t1+ t2a + (n+1)\ t2b + n*t2c + \sum_{i=1}^{n}(t3a+(i+1)*\ t3b+i*\ t3c+i*\ \ t4) \quad +t5$$

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

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

Example:
assume myfunc1: $\theta(n)$
assume myfunc2: $\theta(n^2)$

```
int RandomFunc(int n, int seed)
{
     int x=Rand(seed); //assume Rand function returns a random integer
     for (int i=1; i<=n; i++)
    if (x%2== 0) //if x is even
        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

$O(1)+n*(\max(O(n),O(n^2))+O(1)=O(n^3)$

Lowerbound: take the minimum

$\Omega(1)+n*(\min(\Omega(n), \Omega(n^2))+ \Omega(1)= \Omega(n^2)$

Example: Recursive program to compute n!

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

T(n) = tB + T(n-1)+tA

So the corresponding **recurrence relation** for time complexity is

$$T(n) = \begin{cases} \theta(1) & n=0 \\ T(n-1) + \theta(1) & n \geq 1 \end{cases}$$

$T(n) = T(n-1) + \theta(1)$

$= T(n-2) + 2\theta(1) = \ldots$

$= T(0) + n*\theta(1) \Rightarrow \theta(n)$

Fibonacci numbers:

$$F_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F_{n-1} + F_{n-2} & n \geq 2 \end{cases}$$

i.e. 0,1,1,2,3,4,8,13,21,34…

Example: nonrecursive program to compute Fibonacci numbers

```
int Fibonacci (int n)
{
   int prev2= -1;
   int prev1= 1;
   int sum;
   for (int i = 0; i <= n; ++i)
   {
      sum = prev2+ prev1;
      prev2 = prev1;
      prev1= sum;
   }
   return sum;
}
```

Complexity $O(n)$, $\Omega(n)$, $\theta(n)$

Example: Recursive program to compute Fibonacci numbers

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

Threfore, **recurrence relation** for time complexity is

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

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

UPPER BOUND: O(.)

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

$\leq 2T(n-1) + \theta(1)$

$\leq 2(2T(n-2)) + 3\theta(1) = 2^2 T(n-2) + (2^2-1)\theta(1)$

…

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

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

$\Rightarrow$ T(n) is $O(2^n)$

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

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

LOWER BOUND: $\Omega(.)$

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

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

So T(n) is $\Omega(2^{n/2})$

i.e. Exponential, so infeasible

In fact Fib(n) is $\Omega((3/2)^n)$, however showing it requires more eloborate analysis.