Design and Analysis of Algorithms (UE17CS251)

Unit I - Analysis Framework

Mr. Channa Bankapur
channabankapur@pes.edu

What does it return?

```
foo(n)
  ctr ← 0
  for i ← 1 to n
    ctr ← ctr + 1
  return ctr
```

Return value: …

What does it return?

```
foo(n)
   ctr ← 0
   for i ← 1 to n
      ctr ← ctr + 1
   return ctr
```

Return value: **n**

```
Algorithm
SeqSearch(A[0..n-1], key)
   for i ← 0 to n-1
      if (A[i] = key)
         return i
   return -1
```

How many times “A[i] = key” comparison is made?

What does it return?

```
foo(n)
  ctr ← 0
  for i ← 1 to n
    for j ← 1 to n
        ctr ← ctr + 1
  return ctr
```

Return value: …

What does it return?

```
foo(n)
   ctr ← 0
   for i ← 1 to n
      for j ← 1 to n
            ctr ← ctr
   + 1
   return ctr
```

Return value: **n^2**
(the product rule)

```
BSort(A[0..n-1])
for k ← 0 to n-2
   for i ← 0 to n-2
      if(a[i] > a[i+1])

      Swap(a[i],a[i+1])
```

How many times "A[i] > A[i+1]" comparison is made?

What does it return?

```
foo(n, m)
   ctr ← 0
   for i ← 1 to n
      for j ← 1 to m
            ctr ← ctr + 10
   return ctr
```

Return value: …

What does it return?

```
foo(n, m)
   ctr ← 0
   for i ← 1 to n
      for j ← 1 to m
            ctr ← ctr + 1
   return ctr
```

Return value: **n * m**
(Naive String Matching algorithm's char-to-char comparison is comparable to "ctr ← ctr+1" operation here)

What does it return?

```
foo(n₁, n₂, …, nₘ)
   k ← 0
   for i₁ ← 1 to n₁
      for i₂ ← 1 to n₂
            .
            .
            for iₘ ← 1 to nₘ
                  k ← k + 1
   return k
```

Return value: ...

What does it return?

```
foo(n_1, n_2, …, n_m)
   k ← 0
   for i_1 ← 1 to n_1
      for i_2 ← 1 to n_2
            .
            .
            for i_m ← 1 to n_m
                  k ← k + 1
   return k
```

Return value: $\mathbf{n_1 * n_2 * ... * n_m}$

(the generalized product rule)

What does it return?

```
foo(n)
   ctr ← 0
   for i ← 1 to n-1
      for j ← i+1 to n
            ctr ← ctr + 1
   return ctr
```

Return value: …

What does it return?

```
foo(n)
   ctr ← 0
   for i ← 1 to n - 1
      for j ← i + 1 to n
         ctr ← ctr + 1
   return ctr
```

```
SelectionSort(A[0..n-1])
for i ← 0 to n-2
   min ← i
   for j ← i+1 to n-1
      if(A[j]<A[min])
         min ← j
   Swap (A[i],A[min])
```

How many times "A[j] > A[min]" comparison is made?

Return value: $C(n) = (n-1) + (n-2) + (n-3) + \ldots + 1$

$= \mathbf{n * (n-1) / 2}$

What does it return?

```
foo(n)
    ctr ← 0
    for i ← 1 to n - 1
        for j ← i + 1 to n
            ctr ← ctr + 5
    return ctr
```

Return value:

$C(n) = 5 * ((n-1) + (n-2) + \ldots + 1)$

$= \mathbf{5 * n * (n-1) / 2}$

What does it return?

```
foo(n)
   if(n=1) return 1
   k ← foo(n-1)
   k ← k+1
   k ← k + foo(n-1)
   return k
```

Return value: ...

What does it return?

```
foo(n)
   if(n=1) return 1
   k ← foo(n-1)
   k ← k+1
   k ← k + foo(n-1)
   return k
```

Return value: $\mathbf{2^n - 1}$

```
Hanoi(n, Src, Dest, Int)
   if (n = 0) RETURN
   Hanoi(n-1,Src,Int,Dst)
   Move disk n from S to D
   Hanoi(n-1,Int,Dst,Src)
   RETURN
```

How many times "Move disk" operation is executed?

What does it return?

```
foo(n)
   k ← 0
   i ← n
   while (i > 1)
      k ← k + 1
      i ← ⌊i/2⌋
   return k
```

```
foo(n)
   k ← 0
   if(n > 1)
      k ← foo(⌊n/2⌋)+1
   return k
```

Return value: ...

What does it return?

```
foo(n)
   k ← 0
   i ← n
   while (i > 1)
      k ← k + 1
      i ← ⌊i/2⌋
   return k
```

```
foo(n)
   k ← 0
   if(n > 1)
      k ← foo(⌊n/2⌋)+1
   return k
```

Return value: $C(n) = C(\lfloor n/2 \rfloor) + 1$, $C(1) = 0$

$= \lfloor \mathbf{log_2 n} \rfloor$

What does it return?

```
f(n)
   k ← 0
   if(n > 1)
      k←f(⌊n/2⌋)+1
   return k
```

Return value: …

```
BSch(A[l..r], k)
if(r-l+1 < 1)
      return -1
m = ⌊(l+r) / 2⌋
if(k = A[m])
      return m
else if(k < A[m])
      return BSch(A[l..m-1],k)
else
      return BSch(A[m+1..r],k)
```

How many times "k = A[m]" comparison is made?

What does it return?

```
foo(n)
   k ← 0
   i ← n
   while (i > 1)
      j ← 1
      while (j ≤ n)
            k ← k + 1
            j ← j + 1
      i ← ⌊i/2⌋
   return k
```

Return value: …

What does it return?

```
foo(n)
   k ← 0
   i ← n
   while (i > 1)
      j ← 1
      while (j ≤ n)
            k ← k + 1
            j ← j + 1
      i ← ⌊i/2⌋
   return k
```

Return value: $\mathbf{n * log_2 n}$

(Element-to-element comparison in Merge Sort is comparable to "k←k+1" here)

What does it return?

```
foo(str)
   k ← 0
   n ← length(str)
   for each permutation of str
      j ← 1
      while (j ≤ n)
            k ← k + 1
            j ← j + 1
   return k
```

Return value: …

What does it return?

```
foo(str)
   k ← 0
   n ← length(str)
   for each permn of str
      j ← 1
      while (j ≤ n)
            k ← k + 1
            j ← j + 1
   return k
```

Return value: **n * n!**

Travelling Salesman Problem

```
mincost ← INFINITY
for each perm of n cities
   cost ← 0
   for each edge in the H_ckt
      cost ← cost + edgeCost
   if(cost < mincost)
      mincost ← cost
return mincost
```

How many times "cost + edgeCost" addition is made?

Measuring an Input's Size:

- An algorithm takes same or more time for a larger input of similar kind.
- Algorithm's efficiency is measured as a function of its input size.
- Eg:
    - **n**: size of the array for SequentialSearch(A[0..n-1], K) and SelectionSort(A[0..n-1]).
    - **(m, n, p):** for MatrixMultiplication(A[m,n], B[n,p]) of two matrices of order mxn and nxp.
    - **n (or number of bits used to represent n = $\lfloor \log_2 n \rfloor + 1$)**: the value of the input number in BinaryDigits(n).

Units for Measuring Running Time:

- Count in seconds, minutes, etc. -- Standard unit of time measurement. An algorithm may take different time based on
  - speed of the computing device
  - implementation of the algorithm
  - compiler optimization
- Count the number of times each of the algorithm’s operations is executed.
  - difficult to count each of them
  - not all of them are similar in running time

Count the number of times each of the algorithm's operations is executed.

```
Algorithm SelectionSort(A[0..n-1])
for i ← 0 to n-2
   min ← i
   for j ← i+1 to n-1
      if(A[j] < A[min]) min ← j
   Swap A[i] with A[min]
return A
```

Count the number of times each of the algorithm's operations is executed.

```
Algorithm SelectionSort(A[0..n-1])
i ← 0                                              1
while(i ≤ n-2)                                     1 + n-1
   min ← i                                         n-1
   j ← i+1                                         n-1
   while(j ≤ n-1)                                  n-1 + n(n-1)/2
      if(A[j] < A[min])                            n(n-1)/2
         min ← j                                   n(n-1)/2
      j ← j+1                                      n(n-1)/2
   Swap A[i] with A[min]                           n-1
   i ← i+1                                         n-1
```

Count the number of times each of the algorithm's operations is executed.

Algorithm SelectionSort(A[0..n-1])

i ← 0 $(1) * c_1$

while(i ≤ n-2) $(1 + n-1) * c_2$

min ← i $(n-1) * c_3$

j ← i+1 $(n-1) * c_4$

while(j ≤ n-1) $(n-1 + n(n-1)/2) * c_5$

if(A[j] < A[min]) $(n(n-1)/2) * c_6$

min ← j $(n(n-1)/2) * c_7$

j ← j+1 $(n(n-1)/2)* c$

Count the number of times each of the algorithm's operations is executed.

```
Algorithm SelectionSort(A[0..n-1])
i ← 0                                          (1) *
(c1+c2)
while(i ≤ n-2)
   min ← i                                     (n-1) *
(c2+c3+c4+c5+c9+c10)
   j ← i+1
   while(j ≤ n-1)
      if(A[j] < A[min])                        (n(n-1)/2) *
(c5+c6+c7+c8)
         min ← j
      j ← j+1
   Swap A[i] with A[min]
   i ← i+1
```

$(1) * (c_1+c_2)$

$(n-1) * (c_2+c_3+c_4+c_5+c_9+c_{10})$

$(n(n-1)/2) * (c_5+c_6+c_7+c_8)$

Count the number of times each of the algorithm's operations is executed.

```
Algorithm SelectionSort(A[0..n-1])
i ← 0
```

$(1) * c_{11}$

```
while(i ≤ n-2)
   min ← i
```

$(n-1) * (c_{12})$

```
   j ← i+1
   while(j ≤ n-1)
      if(A[j] < A[min])
```

$(n(n-1)/2) * (c_{13})$

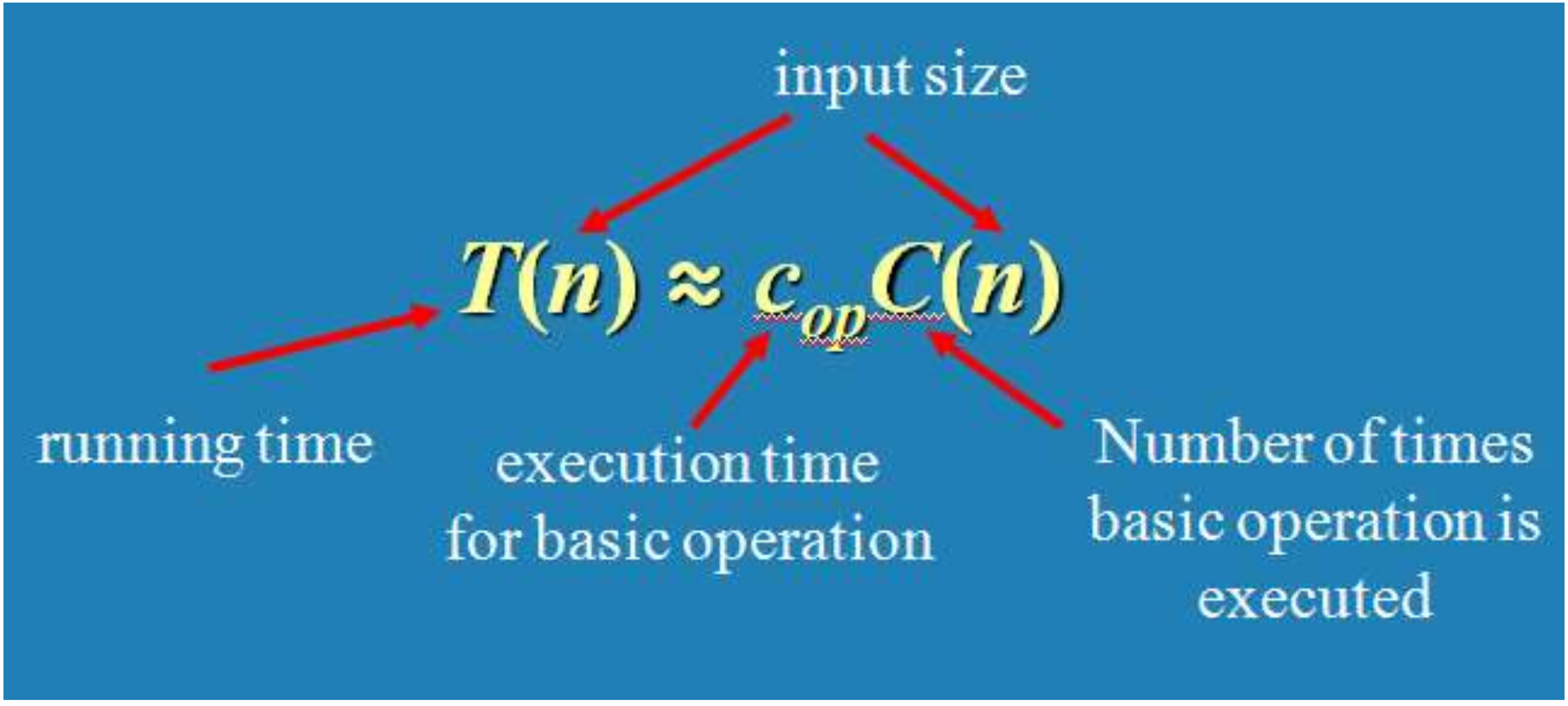
```
         min ← j
      j ← j+1
   Swap A[i] with A[min]
   i ← i+1
```

Count the number of times each of the algorithm's operations is executed.

```
Algorithm SelectionSort(A[0..n-1])
i ← 0                                                    ≤
(n(n-1)/2) * c11
while(i ≤ n-2)
   min ← i                                            ≤ (n(n-
1)/2) * (c12)
   j ← i+1
   while(j ≤ n-1)
      if(A[j] < A[min])                          = (n(n-1)/2) *
(c13)
         min ← j
      j ← j+1
   Swap A[i] with A[min]
   i ← i+1
```

Basic Operation: The most important operation of the algorithm, which is contributing the most to the total running time.

Time efficiency: Counting the number of times the algorithm's **basic operation** is executed on inputs of **size n**.



$\mathbf{T(n) \cong c_{op} * C(n)}$

How much longer will the **Selection Sort** algorithm run if the input size is doubled or increased 10-fold?

For Selection Sorting algorithm,

$$\begin{aligned} \mathbf{C(n)} \quad &\mathbf{= (n(n-1)/2)} \\ &\mathbf{=} (n^2/2 - n/2) \\ &\cong \mathbf{n^2/2} \text{ for sufficiently large value of n.} \end{aligned}$$

$C(n) \cong n^2/2$ for sufficiently larger value of n.

$$\frac{T(10n)}{T(n)} \cong \frac{c_{op}C(10n)}{c_{op}C(n)} \cong \frac{(1/2)(10n)^2}{(1/2)n^2} = 100$$

For n=1,000 → T(n) ≅ 2 milliseconds

For n=10,000 → T(n) ≅ 200 ms = 0.2 seconds

For n=100k → T(n) ≅ 0.2*100 = 20 seconds

	Selection Sort (Cop = 4 nanosec)			
n	**n^2**	**4 nanosec * (n^2)/2**		
1	**1**	**0.000000002**	**2.00**	**nanosec.**
10	100	2E-07	200.00	nanosec.
100	10,000	2E-05	20.00	microsec.
1E+03	**1E+06**	**2E-03**	**2.00**	**millisec.**
1E+04	1E+08	2E-01	200.00	millisec.
1E+05	1E+10	2E+01	20.00	sec.
1E+06	**1E+12**	**2E+03**	**33.33**	**min.**
1E+07	1E+14	2E+05	2.31	days.
1E+08	1E+16	2E+07	231.48	days.
1E+09	**1E+18**	**2E+09**	**63.42**	**yrs.**

Analysis Framework ignores multiplicative constants and concentrates on the **basic operation count's order of growth** to within a constant multiple for large-size inputs. So, the order of growth of selection sort is primarily driven by "$\mathbf{n^2}$".

In the same way, let the order of growth of Mergesort is driven by "**n log n**". Suppose, the cost of the basic operation of the selection sort is 2 nanoseconds and that of the mergesort is 40 nanoseconds. What could be the estimated execution time of mergesort on an input size of:

(i) 1 million?

(ii) 1 billion?

	Merge Sort (Cop = 40 nanosec)			
n	**n logn**	**40 nanosec * n logn**		
1	**1**	**0.00000004**	**40.00**	**nanosec.**
10	33	1.33E-06	1.33	microsec.
100	664	2.66E-05	26.58	microsec.
1E+03	**9.97E+03**	**3.99E-04**	**398.63**	**microsec.**
1E+04	1.33E+05	5.32E-03	5.32	millisec.
1E+05	1.66E+06	6.64E-02	66.44	millisec.
1E+06	**1.99E+07**	**7.97E-01**	**797.26**	**millisec.**
1E+07	2.33E+08	9.30E+00	9.30	sec.
1E+08	2.66E+09	1.06E+02	1.77	min.
1E+09	**2.99E+10**	**1.20E+03**	**19.93**	**min.**

	input size	Selection Sort		Merge Sort	Quick Sort	
log n	n	n(n-1)/2	n^2	n + 2n log n	2n - 1 + n log n	n log n
1	2	1	4	6	5	2
2	4	6	16	20	15	8
3	8	28	64	56	39	24
4	16	120	256	144	95	64
5	32	496	1,024	352	223	160
6	64	2,016	4,096	832	511	384
7	128	8,128	16,384	1,920	1,151	896
8	256	32,640	65,536	4,352	2,559	2,048
9	512	130,816	262,144	9,728	5,631	4,608
10	1,024	523,776	1,048,576	21,504	12,287	10,240
11	2,048	2,096,128	4,194,304	47,104	26,623	22,528
12	4,096	8,386,560	16,777,216	102,400	57,343	49,152
13	8,192	33,550,336	67,108,864	221,184	122,879	106,496
14	16,384	134,209,536	268,435,456	475,136	262,143	229,376
15	32,768	536,854,528	1,073,741,824	1,015,808	557,055	491,520
16	65,536	2,147,450,880	4,294,967,296	2,162,688	1,179,647	1,048,576
17	131,072	8,589,869,056	17,179,869,184	4,587,520	2,490,367	2,228,224
18	262,144	34,359,607,296	68,719,476,736	9,699,328	5,242,879	4,718,592
19	524,288	137,438,691,328	274,877,906,944	20,447,232	11,010,047	9,961,472
20	1,048,576	549,755,289,600	1,099,511,627,776	42,991,616	23,068,671	20,971,520

log n	n	n log n	10 n log n	0.1 * n^2	n^2
1	2	2	20	0	4
2	4	8	80	2	16
3	8	24	240	6	64
4	16	64	640	26	256
5	32	160	1,600	102	1,024
6	64	384	3,840	410	4,096
7	128	896	8,960	1,638	16,384
8	256	2,048	20,480	6,554	65,536
9	512	4,608	46,080	26,214	262,144
10	1,024	10,240	**102,400**	**104,858**	1,048,576
11	2,048	22,528	225,280	419,430	4,194,304
12	4,096	49,152	491,520	1,677,722	16,777,216
13	8,192	106,496	1,064,960	6,710,886	67,108,864
14	16,384	229,376	2,293,760	26,843,546	268,435,456
15	32,768	491,520	4,915,200	107,374,182	1,073,741,824
16	65,536	1,048,576	10,485,760	429,496,730	4,294,967,296
17	131,072	2,228,224	22,282,240	1,717,986,918	17,179,869,184
18	262,144	4,718,592	47,185,920	6,871,947,674	68,719,476,736
19	524,288	9,961,472	99,614,720	27,487,790,694	274,877,906,944
20	1,048,576	20,971,520	209,715,200	1.09951E+11	1.09951E+12

What if the basic operation runs for 2^{100} times?

Suppose,

- An algorithm takes 2^{100} operations.
  - Tower of Hanoi takes about 2^{100} operations for 100 disks.
- Each operation takes just one clock tick.
- 1 terahertz processor exists. ($10^{12} \cong 2^{40}$)

So, the algorithm takes $2^{100} / 2^{40} = 2^{60}$ seconds.

$\cong 2^{35}$ years ($\because$ 1 year = 60*60*24*365 $\cong 2^{25}$ seconds)

$\cong 32 * 10^9$ years ($\because 10^9 \cong 2^{30}$)

= 32 billion years

BTW, it's estimated that the Earth was formed about 4.5 billion years back and the big bang happened about 14 billion yrs back!

Algorithms that require an **exponential number** of operations are practical for solving only problems with very small input sizes.

log n	n	n^3	100 n^3	1.01 ^ n	2^n
1	2	8	800	1	4
2	4	64	6,400	1	16
3	8	512	51,200	1	256
4	16	4,096	409,600	1	65,536
5	32	32,768	3,276,800	1	4,294,967,296
6	64	262,144	26,214,400	2	1.84467E+19
7	128	2,097,152	209,715,200	4	3.40282E+38
8	256	16,777,216	1,677,721,600	13	1.15792E+77
9	512	134,217,728	13,421,772,800	163	1.3408E+154
10	1,024	1,073,741,824	107,374,182,400	26,613	#NUM!
11	2,048	8,589,934,592	858,993,459,200	708,228,675	#NUM!
12	4,096	68,719,476,736	6.87195E+12	5.01588E+17	#NUM!
13	8,192	549,755,813,888	5.49756E+13	2.5159E+35	#NUM!
14	16,384	4.39805E+12	4.39805E+14	6.32977E+70	#NUM!
15	32,768	3.51844E+13	3.51844E+15	4.0066E+141	#NUM!
16	65,536	2.81475E+14	2.81475E+16	1.6053E+283	#NUM!
17	131,072	2.2518E+15	2.2518E+17	#NUM!	#NUM!
18	262,144	1.80144E+16	1.80144E+18	#NUM!	#NUM!
19	524,288	1.44115E+17	1.44115E+19	#NUM!	#NUM!
20	1,048,576	1.15292E+18	1.15292E+20	#NUM!	#NUM!

A Problem can be:

- Non-computable
  - No solution exists theoretically
- Computable and Intractable
  - Solution exists, but …
  - exponential time (aka super-polynomial time)
- Computable and Tractable
  - polynomial time

Orders of growth:

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

log n	n	n log n	n^2	n^3	2^n	n!
1	2	2	4	8	4	2
2	4	8	16	64	16	24
3	8	24	64	512	256	40320
4	16	64	256	4,096	65,536	2.09228E+13
5	32	160	1,024	32,768	4,294,967,296	2.63131E+35
6	64	384	4,096	262,144	1.84467E+19	1.26887E+89
7	128	896	16,384	2,097,152	3.40282E+38	3.8562E+215
8	256	2,048	65,536	16,777,216	1.15792E+77	#NUM!
9	512	4,608	262,144	134,217,728	1.3408E+154	#NUM!
10	1,024	10,240	1,048,576	1,073,741,824	#NUM!	#NUM!
11	2,048	22,528	4,194,304	8,589,934,592	#NUM!	#NUM!
12	4,096	49,152	16,777,216	68,719,476,736	#NUM!	#NUM!
13	8,192	106,496	67,108,864	549,755,813,888	#NUM!	#NUM!
14	16,384	229,376	268,435,456	4.39805E+12	#NUM!	#NUM!
15	32,768	491,520	1,073,741,824	3.51844E+13	#NUM!	#NUM!
16	65,536	1,048,576	4,294,967,296	2.81475E+14	#NUM!	#NUM!
17	131,072	2,228,224	17,179,869,184	2.2518E+15	#NUM!	#NUM!
18	262,144	4,718,592	68,719,476,736	1.80144E+16	#NUM!	#NUM!
19	524,288	9,961,472	274,877,906,944	1.44115E+17	#NUM!	#NUM!
20	1,048,576	20,971,520	1.09951E+12	1.15292E+18	#NUM!	#NUM!

log n	√n	n	n log n	n * n^¼	n √n	n^2
1	1	2	2	2	3	4
2	2	4	8	6	8	16
3	3	8	24	13	23	64
4	4	16	64	32	64	256
5	6	32	160	76	181	1,024
6	8	64	384	181	512	4,096
7	11	128	896	431	1,448	16,384
8	16	256	2,048	1,024	4,096	65,536
9	23	512	4,608	2,435	11,585	262,144
10	32	1,024	10,240	5,793	32,768	1,048,576
11	45	2,048	22,528	13,777	92,682	4,194,304
12	64	4,096	49,152	32,768	262,144	16,777,216
13	91	8,192	106,496	77,936	741,455	67,108,864
14	128	16,384	229,376	185,364	2,097,152	268,435,456
15	181	32,768	491,520	440,872	5,931,642	1,073,741,824
16	256	65,536	**1,048,576**	**1,048,576**	16,777,216	4,294,967,296
17	362	131,072	2,228,224	2,493,948	47,453,133	17,179,869,184
18	512	262,144	4,718,592	5,931,642	134,217,728	68,719,476,736
19	724	524,288	9,961,472	14,107,901	379,625,062	274,877,906,944
20	1,024	1,048,576	20,971,520	33,554,432	1,073,741,824	1.09951E+12

There are many algorithms for which running time depends not only on an **input size** but also on the **specifics of a particular input**.

```
Algorithm SequentialSearch(A[0..n-1], K)
    //Outputs the index of the first element of A that
    // matches K or -1 if there are no matching elements.
    i ← 0
    while (i < n) and (A[i] ≠ K) do
        i ← i + 1
    if (i < n) return i
    return -1
```

```
Algorithm SequentialSearch(A[0..n-1], K)
   //Outputs the index of the first element of A that
   // matches K or -1 if there are no matching elements.
   i ← 0
   while (i < n) and (A[i] ≠ K) do
      i ← i + 1
   if (i < n) return i
   return -1
```

Input size: n.

Basic Operation: **(i < n) and (A[i] ≠ K)**

$C_{worst}(n) = n+1$

$C_{best}(n) = 1$

$C_{avg}(n) = ?$

Worst case: $C_{worst}(n)$ – maximum over inputs of size *n*

Best case: $C_{best}(n)$ – minimum over inputs of size *n*

Average case: $C_{avg}(n)$ – “average” over inputs of size *n*

- Number of times the basic operation will be executed on typical input
- NOT the average of worst and best case

Amortized efficiency (out of syllabus)

`Algorithm SequentialSearch(A[0..n-1], K)`

If the search key is certainly present in the array:

$C_{avg,\ key\ present}(n) = (1 + 2 + \ldots + n) / n$

$C_{avg,\ key\ present}(n) = (n + 1) / 2$

$C_{avg,\ key\ absent}(n) = (n + 1)$

Let **'p'** be the probability of the search key present in the array.

$\mathbf{C_{avg}(n) = p}(n + 1) / 2 \mathbf{+ (1 - p)}(n + 1)$

When $p = 1$, $C_{avg}(n) = (n + 1) / 2$

When $p = 0$, $C_{avg}(n) = (n + 1)$

When $p = 0.5$, $C_{avg}(n) = 0.75 * (n + 1)$

`Algorithm SequentialSearch(A[0..n-1], K)`

Input Size: **`n`**

Basic Operation : **`(i < n) and (A[i] ≠ K)`**

$C_{worst}(n)$ = Count of the basic operation at the max

= **n + 1**

$C_{best}(n)$ = **1**

$C_{avg}(n)$ = from **(n+1)/2** to **(n+1)** depending on the probability of search key being present in the input array.

Time efficiency analysis framework concentrates on the order of growth of the **basic operation count** of an algorithm as the principal indicator of the algorithm's efficiency.

Asymptotic Notations:

- **"Big Oh" O(*g*(*n*))**:

  class of functions $t(n)$ that grow **no faster** than $g(n)$

- **"Big Omega" Ω(*g*(*n*))**:

  class of functions $t(n)$ that grow **at least as fast** as $g(n)$

- **"Big Theta" Θ(*g*(*n*))**:

  class of functions $t(n)$ that grow **at the same rate** as $g(n)$

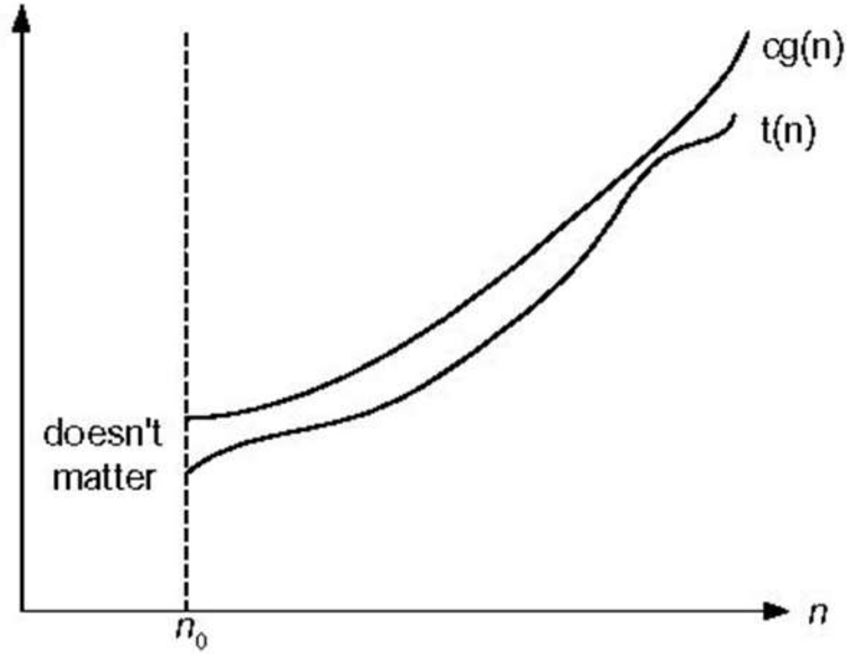


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

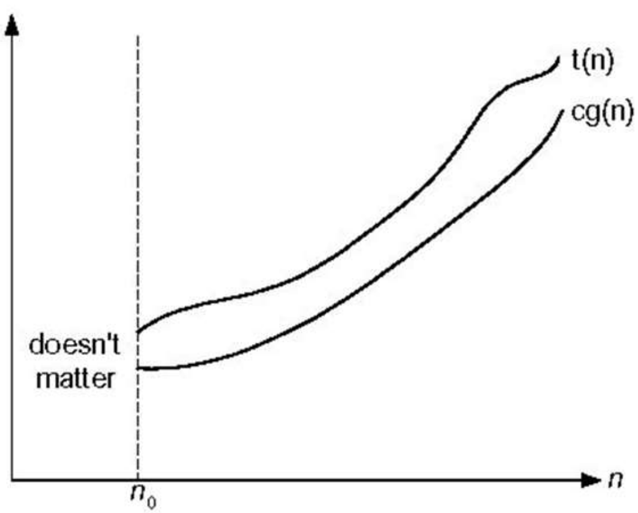


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

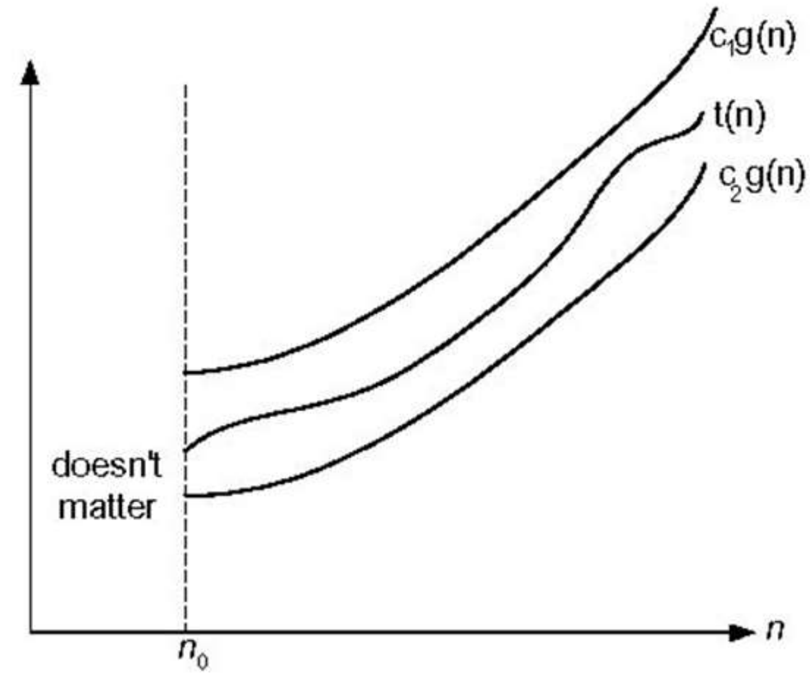


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

"Big Oh" O(g(n)):

A function t(n) is said to be in **O(**g(n)**)** if t(n) is bounded above by some constant multiple of g(n) for all large n.

$\mathbf{t(n) \le cg(n)\ \forall\ n \ge n_0}$

Denoted as $\mathbf{t(n) \in O(g(n))}$

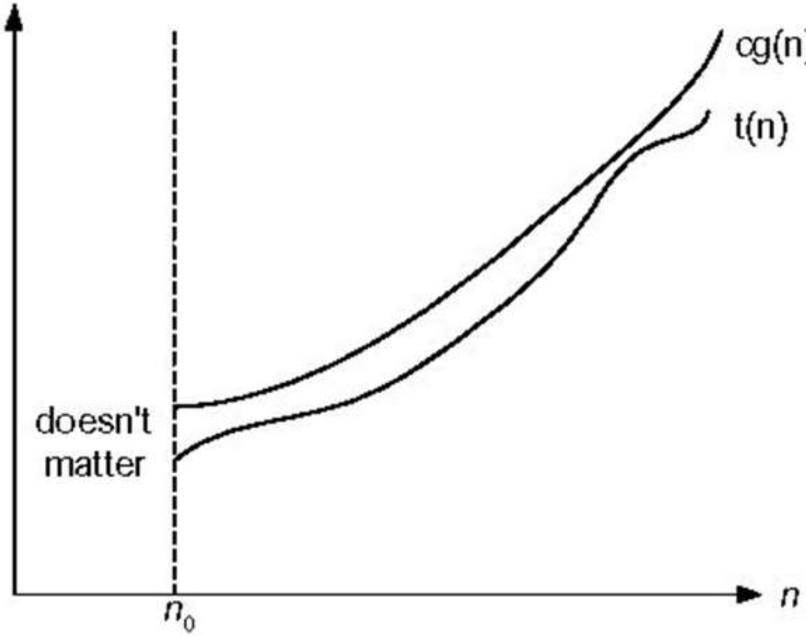


Figure 2.1 Big-oh notation: $t(n) \in O(g(n))$

Eg: $100n+5 \in O(n)$

$100n+5 \le 100n+5n$ ($\forall$ $n \ge 1$)

$100n+5 \le 105n\ \forall\ n \ge 1$ ($\therefore$ $\mathbf{c=105, n_0=1}$)

$100n+5 \le 100n+n$ ($\forall$ $n \ge 5$) $= 101n\ \forall\ n \ge 5$ ($\therefore$ $\mathbf{c=101, n_0=5}$)

Eg: $100n+5 \in O(n^2)$

Eg: $n(n-1)/2 \in O(n^2)$

“Big Omega” Ω(g(n)):

A function t(n) is said to be in **Ω(**g(n)**)** if t(n) is bounded below by some constant multiple of g(n) for all large n.

$t(n) \geq cg(n)\ \forall\ n \geq n_0$

Denoted by **$t(n) \in \Omega(g(n))$**

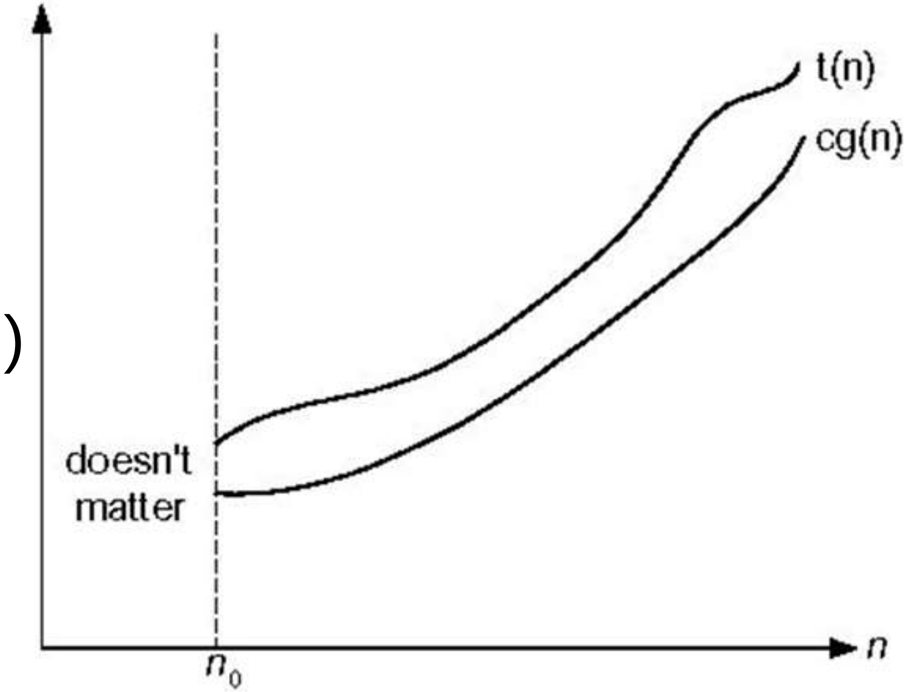


Fig. 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Eg: $100n+5 \in \mathbf{\Omega}(n)$

$100n+5 \geq 100n$ ($\forall$ $n \geq 0$)

$100n+5 \geq 100n\ \forall\ n \geq 0$ ($\therefore$ **$c=100$, $n_0=0$**)

Eg: $n(n-1)/2 \in \mathbf{\Omega}(n^2)$

Eg: $n(n-1)/2 \in \mathbf{\Omega}(n)$

“Big Theta” Θ(g(n)):

A function t(n) is said to be in **Θ(**g(n)**)** if t(n) is bounded both above and below by some constant multiples of g(n) for all large n.

$\mathbf{c_2 g(n) \leq t(n) \leq c_1 g(n)\ \forall\ n \geq n_0}$

Denoted by **t(n) ∈ Θ(g(n))**

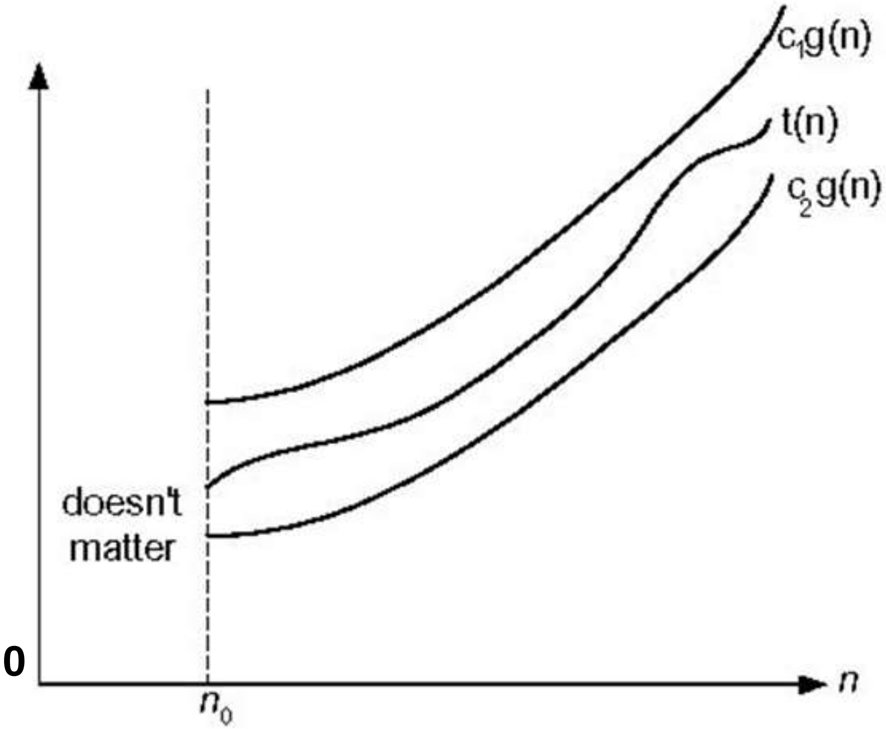


Figure 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Eg: n(n-1)/2 ∈ **Θ**(n^2)

$n(n-1)/2 = n^2/2 - n/2 \leq n^2/2\ \forall\ n \geq 0$

$n(n-1)/2 = n^2/2 - n/2 \geq n^2/2 - n^2/4 = n^2/4\ \forall\ n \geq 2$

$\mathbf{n^2/4 \leq n(n-1)/2 \leq n^2/2}\ \forall\ n \geq 2$ (∴ $\mathbf{c_1=1/2,\ c_2=1/4,\ n_0=2}$)

$t(n) \in O(g(n))$

Nonnegative functions defined on the set of natural numbers

- $t(n)$: algorithm's running time by counting basic operation
- $g(n)$: simple function to compare the count with.

$O(g(n))$ is the set of all functions with a **smaller or same** order of growth as $g(n)$.

E.g.: $100n + 5 \in O(n)$

$\in O(n^2)$

$\notin O(\log n)$

$n(n-1)/2 \in O(n^2)$

$\in O(n^{10})$

$\notin O(n)$

Ω(g(n)) is the set of all functions with a **larger or same** order of growth as g(n).

E.g.: $100n + 5 \in \mathbf{\Omega}(n)$

$\notin \mathbf{\Omega}(n^2)$

$\in \mathbf{\Omega}(\log n)$

$n(n-1)/2 \in \mathbf{\Omega}(n^2)$

$\notin \mathbf{\Omega}(n^{10})$

$\in \mathbf{\Omega}(n)$

Θ(g(n)) is the set of all functions that have the **same order** of growth as g(n).

E.g.: $n(n-1)/2 \in \mathbf{\Theta}(n^2)$

$\notin \mathbf{\Theta}(n^3)$

$\notin \mathbf{\Theta}(n \log n)$

- $\mathbf{t(n) \in O(t(n))}$

- $\mathbf{t(n) \in O(g(n))}$ iff $\mathbf{g(n) \in \Omega(t(n))}$

  $\mathbf{t(n) \in \Theta(g(n))}$ iff $\mathbf{g(n) \in \Theta(t(n))}$

- If $\mathbf{t(n) \in O(g(n))}$ and $\mathbf{g(n) \in O(h(n))}$, then $\mathbf{t(n) \in O(h(n))}$

- If $\mathbf{t_1(n) \in O(g_1(n))}$ and $\mathbf{t_2(n) \in O(g_2(n))}$, then
  $\mathbf{t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})}$

Theorem: If $\mathbf{t_1(n) \in O(g_1(n))}$ and $\mathbf{t_2(n) \in O(g_2(n))}$, then

$$\mathbf{t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})}$$

That is, if

$t_1(n) \le c_1 g_1(n)\ \forall\ n \ge n_1$ **and** $t_2(n) \le c_2 g_2(n)\ \forall\ n \ge n_2$

then

$t_1(n) + t_2(n) \le c \max\{g_1(n), g_2(n)\}\ \forall\ n \ge n_0$

Proof:

Hint:

W.k.t. for real numbers a_1, b_1, a_2, b_2

if $a_1 \le b_1$ and $a_2 \le b_2$, then $a_1 + a_2 \le 2 \max\{b_1, b_2\}$

Theorem: If $\mathbf{t_1(n) \in O(g_1(n))}$ and $\mathbf{t_2(n) \in O(g_2(n))}$, then

$$\mathbf{t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})}$$

Proof:

$t_1(n) \le c_1 g_1(n)\ \forall\ n \ge n_1$

$t_2(n) \le c_2 g_2(n)\ \forall\ n \ge n_2$

$t_1(n) + t_2(n) \quad \le c_1 g_1(n) + c_2 g_2(n)\ \forall\ n \ge \max\{n_1, n_2\}$

$\le c_3 g_1(n) + c_3 g_2(n)$ where $c_3 = \max\{c_1, c_2\}$

$\le c_3\ (g_1(n) + g_2(n))$

$\le c_3\ 2 \max\{g_1(n), g_2(n)\}$

Therefore, $\mathbf{t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})}$

where $c = 2 \max\{c_1, c_2\}$ and $n_0 = \max\{n_1, n_2\}$

Limits are useful for comparing orders of growth of two specific functions.

Limit of the ratio of two functions reveals the relative orders of growth of the two functions.

$$\lim_{n\to\infty} T(n)/g(n) = \begin{cases} 0 & \text{order of growth of } T(n) < \text{order of growth of } g(n) \\ c>0 & \text{order of growth of } T(n) = \text{order of growth of } g(n) \\ \infty & \text{order of growth of } T(n) > \text{order of growth of } g(n) \end{cases}$$

How can we relate these three cases to
T(n) ∈ **O**(g(n)), T(n) ∈ **Ω**(g(n)) and T(n) ∈ **Θ**(g(n)) ?
T(n) ∈ **o**(g(n)), T(n) ∈ **ω**(g(n)) ?

$$\lim_{n\to\infty} T(n)/g(n) = \begin{cases} 0 & \text{order of growth of } T(n) < \text{ order of growth of } g(n) \\ c>0 & \text{order of growth of } T(n) = \text{ order of growth of } g(n) \\ \infty & \text{order of growth of } T(n) > \text{ order of growth of } g(n) \end{cases}$$

E.g.: Compare the orders of growth of **n(n-1)/2** and **n²**

$$\lim_{n\to\infty} \frac{\frac{1}{2}n(n-1)}{n^2} = \frac{1}{2}\lim_{n\to\infty}\frac{n^2-n}{n^2} = \frac{1}{2}\lim_{n\to\infty}\left(1-\frac{1}{n}\right) = \frac{1}{2}$$

$$\frac{1}{2}n(n-1) \in \Theta(n^2)$$

L'Hôpital's rule: If $lim_{n\to\infty} f(n) = lim_{n\to\infty} g(n) = \infty$ and the derivatives f', g' exist, then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$$

E.g.: Compare the orders of growth of $\mathbf{log_2\ n}$ and $\sqrt{(\mathbf{n})}$

$$\lim_{n\to\infty} \frac{\log_2 n}{\sqrt{n}} = \lim_{n\to\infty} \frac{(\log_2 n)'}{(\sqrt{n})'} = \lim_{n\to\infty} \frac{(\log_2 e)\,\frac{1}{n}}{\frac{1}{2\sqrt{n}}} = 2\log_2 e \lim_{n\to\infty} \frac{1}{\sqrt{n}} = 0$$

$\log_2 n \in o(\sqrt{n})$ Little-oh notation

Stirling's formula: $n! \approx (2\pi n)^{1/2}(n/e)^n$

E.g.: Compare the orders of growth of **n!** and **2^n**.

$$\lim_{n\to\infty}\frac{n!}{2^n}=\lim_{n\to\infty}\frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{2^n}=\lim_{n\to\infty}\sqrt{2\pi n}\frac{n^n}{2^n e^n}=\lim_{n\to\infty}\sqrt{2\pi n}\left(\frac{n}{2e}\right)^n=\infty$$

$$n!\in\Omega(2^n)$$

- All logarithmic functions $\mathbf{log_a\, n}$ belong to the same class $\mathbf{\Theta(log\, n)}$ no matter what base of the logarithm $\mathbf{a > 1}$ is.
  - $\mathbf{log_{10}\, n \in \Theta(log_2\, n)}$

- All polynomials of the same degree **k** belong to the same class: $\mathbf{a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0 \in \Theta(n^k)}$

- Exponential functions $\mathbf{a^n}$ have different orders of growth for different **a**’s.
  - $\mathbf{3^n \notin \Theta(2^n)}$

Basic asymptotic efficiency classes

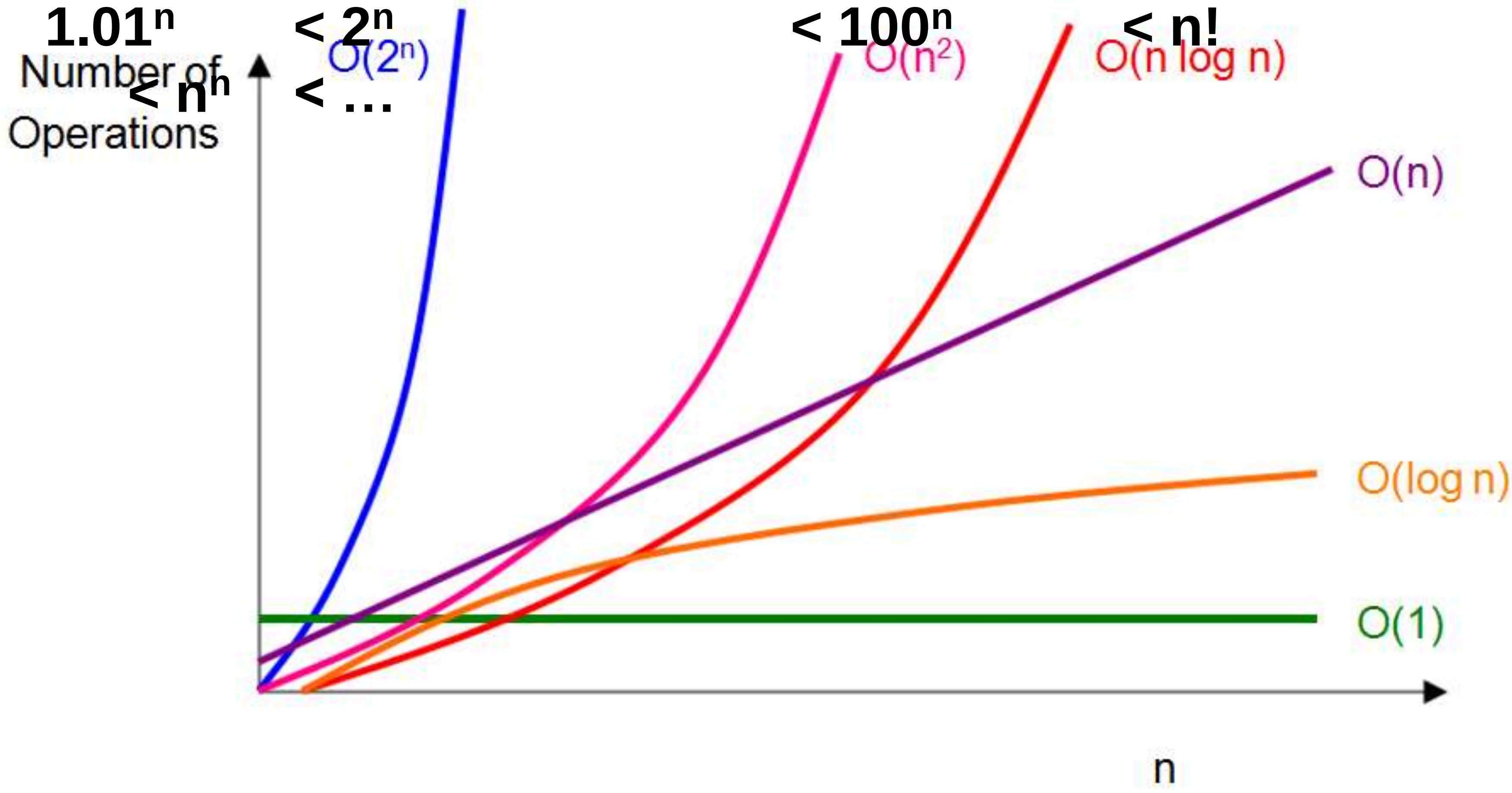
Class	Name
1	*constant*
$\log n$	*logarithmic*
n	*linear*
$n \log n$	*linearithmic*

Class	Name
n^2	*quadratic*
n^3	*cubic*
2^n	*exponential*
$n!$	*factorial*

$1 < \log\log n < \log n < n^{0.001} < n^{0.5} <$

$n < n \log n < n^2 < n^3 < n^{100} <$

$1.01^n < 2^n < 100^n < n! < n^n < \ldots$



Analysing time efficiency of **recursive/non-recursive** algorithms:

1. input size?

1. basic operation?

1. $C_{best}(n)$, $C_{worst}(n)$ and $C_{avg}(n)$, or just $C(n)$?

1. Closed-form formula for C(n)
   If C(n) is a recurrence, solve the recurrence (or, at the very least, establish its solution's order of growth) by backward substitutions or some other method.

1. $T(n) \in$ **O(), Ω(), Θ()?**

```
Algorithm SequentialSearch(A[0..n-1], K)
    //Outputs the index of the first element of A that
    // matches K or -1 if there are no matching elements.
    i ← 0
    while (i < n) and (A[i] ≠ K) do
        i ← i + 1
    if (i < n) return i
    return -1
```

Input size: n.

Basic Operation: **(i < n) and (A[i] ≠ K)**

$\mathbf{C_{worst}(n) = n+1} \in \mathbf{\Theta(n)}$

$\mathbf{C_{best}(n) = 1} \in \mathbf{\Theta(1)}$

$\mathbf{C_{avg}(n) =}$ from $\mathbf{(n+1)/2}$ to $\mathbf{(n+1)} \in \mathbf{\Theta(n)}$

ALGORITHM *MaxElement*$(A[0..n-1])$

//Determines the value of the largest element in a given array
//Input: An array $A[0..n-1]$ of real numbers
//Output: The value of the largest element in A
$maxval \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
    if $A[i] > maxval$
        $maxval \leftarrow A[i]$
return $maxval$

Input Size: **`n`**

Basic Operation : **`( A[i] > maxval )`**

$\mathbf{C_{worst}(n) = C_{best}(n) = n\text{-}1 \in \Theta(n)}$

ALGORITHM *MatrixMultiplication*($A[0..n-1, 0..n-1]$, $B[0..n-1, 0..n-1]$)

//Multiplies two n-by-n matrices by the definition-based algorithm
//Input: Two n-by-n matrices A and B
//Output: Matrix $C = AB$
for $i \leftarrow 0$ **to** $n-1$ **do**
    for $j \leftarrow 0$ **to** $n-1$ **do**
        $C[i, j] \leftarrow 0.0$
        for $k \leftarrow 0$ **to** $n-1$ **do**
            $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$
return C

Input Size: **`n`**

Basic Operation : Multiplication **`(A[i,k]*B[k,j])`**

$\mathbf{C_{worst}(n) = C_{best}(n) = n^3 \in \Theta(n^3)}$

Array with distinct elements:

For every distinct pair of elements (A_i , A_j), A_i != A_j.

```
Algorithm UniqueElements(A[0..n-1])
//Determines whether all the elements in a given are distinct.
//Input: An array A[0..n-1]
//Output: Returns "true" if all the elements in A are distinct
// and "false" otherwise.
   for i ← 0 n-2 do
      for j ← i+1 to n-1
         if(A[i] = A[j]) return false
   return true
```

`Algorithm: UniqueElements(A[0..n-1])`

Input Size: **`n`**

Basic Operation : (**`A[i] = A[j]`**)

C_{worst}(n) = n * (n - 1) / 2 $\in$ Θ(n^2)

C_{best}(n) = 1 $\in$ Θ(1)

$$
\begin{aligned}
C_{worst}(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1)-(i+1)+1] = \sum_{i=0}^{n-2} (n-1-i) \\
&= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2} \\
&= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2).
\end{aligned}
$$

Number of bits needed to represent decimal value n:

ALGORITHM *Binary*(n)

//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
$count \leftarrow 1$
while $n > 1$ **do**
    $count \leftarrow count + 1$
    $n \leftarrow \lfloor n/2 \rfloor$
return $count$

ALGORITHM *BinRec*(n)

//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
if $n = 1$ **return** 1
else return $BinRec(\lfloor n/2 \rfloor) + 1$

`Algorithm: BinaryDigits(n)`

Input Size: **`n`**

Basic Operation : **`addition`**

$\mathbf{C_{worst}(n) = C_{best}(n)}$

$C(n) \quad = C(n/2) + 1,\ C(1) = 0$

$\qquad\qquad = C(n/4) + 2$

$\qquad\qquad = C(n/2^3) + 3$

$\qquad\qquad = C(n/2^i) + i$

$C(n/2^i)$ is $C(1)$ when $n/2^i = 1 \Rightarrow n = 2^i \Rightarrow i = \log_2 n$

$C(n) \quad = \mathbf{\log_2 n} \in \mathbf{\Theta(\log n)}$

$F(n) = F(n-1) \cdot n \quad \text{for } n > 0,$

ALGORITHM $F(n)$

//Computes $n!$ recursively
//Input: A nonnegative integer n
//Output: The value of $n!$
if $n = 0$ **return** 1
else return $F(n-1) * n$

Input Size: **`n`**

Basic Operation : **`multiplication`**

C(n) = C(n - 1) + 1, C(0) = 0
= **n** ∈ **Θ(n)**

Basic Operation : **`(n = 0)`** **i.e. the number calls to F(n)**

C(n) = C(n - 1) + 1, C(0) = 1
= **n + 1** ∈ **Θ(n)**

```
Algorithm TowerOfHanoi(n, Src, Aux, Dst)
   if (n = 0) RETURN
   Hanoi(n-1, Src, Dst, Aux)
   Move disk n from Src to Dst
   Hanoi(n-1, Aux, Src, Dst)
```

Input Size: **`n`**

Basic Operation : **`Move disk n from Src to Dst`**

$C(n) = \mathbf{2C(n-1) + 1}$ for $n > 0$ and $C(0)=0$

$C(n) = 2\,[2\,C(n-2) + 1] + 1 \quad = 2^2\,C(n-2) + 2 + 1$

$= 2^2\,[2\,C(n-3) + 1]+2+1 = 2^3\,C(n-3) + 2^2 + 2^1 + 2^0$

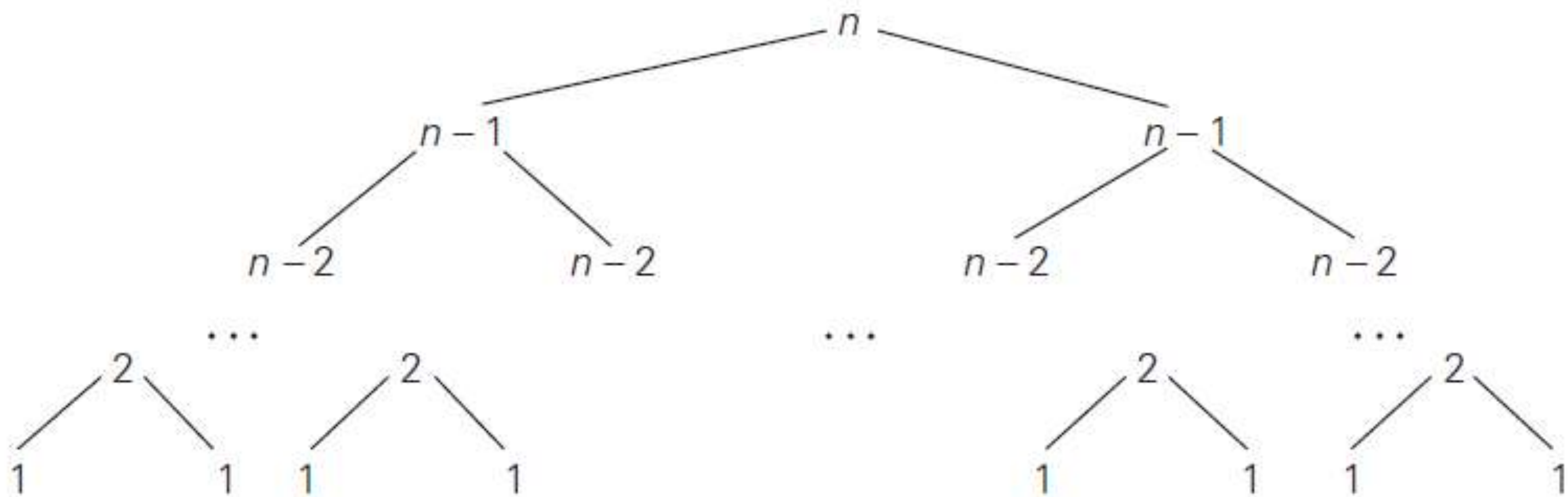
$= 2^3\,C(n-3) + 2^3 - 1$

$= 2^i\,C(n-i) + 2^i - 1$

$C(n-i)$ becomes $C(0)$ when $n-i = 0 \Rightarrow i = n$

$C(n) = 2^n\,C(n-n) + 2^n - 1 \;= \mathbf{2^n - 1 \in \Theta(2^n)}$

Tree of recursive calls made by the recursive algorithm for the Tower of Hanoi puzzle.



$$C(n) = \sum_{l=0}^{n-1} 2^l \text{ (where } l \text{ is the level in the tree in Figure 2.5)} = 2^n - 1$$

```
Algorithm TowerOfHanoi(n, Src, Aux, Dst)
   if (n = 0) RETURN
   Hanoi(n-1, Src, Dst, Aux)
   Move disk n from Src to Dst
   Hanoi(n-1, Aux, Src, Dst)
```

Input Size: **`n`**

Basic Operation : **`Move disk n from Src to Dst`**

$C(n) \quad = 2\,C(n-1) + 1,\ C(0) = 0$

$\qquad\quad = \mathbf{2^n - 1} \in \mathbf{\Theta(2^n)}$

Basic Operation : **`(n = 0)`**

$C(n) \quad = 2\,C(n-1) + 1,\ C(0) = 1$

$\qquad\quad = \mathbf{2^n + 2^n - 1} \in \mathbf{\Theta(2^n)}$

```
Algorithm TowerOfHanoi2(n, A, B, C)
   if (n = 1)
      Move disk n from A to C
      RETURN
   TowerOfHanoi2(n-1, A, C, B)
   Move disk n from A to C
   TowerOfHanoi2(n-1, B, A, C)
   RETURN
```

Basic Operation : **`Move disk n from A to C`**

C(n) = 2 * C(n - 1) + 1, C(1) = 1

= $\mathbf{2^n - 1} \in \mathbf{\Theta(2^n)}$

Basic Operation : **`(n=1)`** **i.e. the number of function calls**

C(n) = 2 * C(n - 1) + 1, **C(1) = 1**

= $\mathbf{2^n - 1} \in \mathbf{\Theta(2^n)}$

```
Algorithm TowerOfHanoi3(n, A, B, C)
   if (n = 0) RETURN
   if (n>1) TowerOfHanoi3(n-1, A, C, B)
   Move disk n from A to C
   if (n>1) TowerOfHanoi3(n-1, B, A, C)
   RETURN
```

Basic Operation : **`Move disk n from A to C`**

C(n) = 2 * C(n - 1) + 1, C(1) = 1

$= \mathbf{2^n - 1} \in \mathbf{\Theta(2^n)}$

Basic Operation : **`(n=0)`** **i.e. the number of function calls**

C(n) = 2 * C(n - 1) + 1, **C(0) = C(1) = 1**

$= \mathbf{2^n - 1} \in \mathbf{\Theta(2^n)}$

$$0,\ 1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34, \ldots$$

$$F(n) = F(n-1) + F(n-2) \quad \text{for } n > 1 \qquad F(0) = 0, \qquad F(1) = 1$$

ALGORITHM $F(n)$

//Computes the nth Fibonacci number recursively by using its definition
//Input: A nonnegative integer n
//Output: The nth Fibonacci number
if $n \leq 1$ **return** n
else return $F(n-1) + F(n-2)$

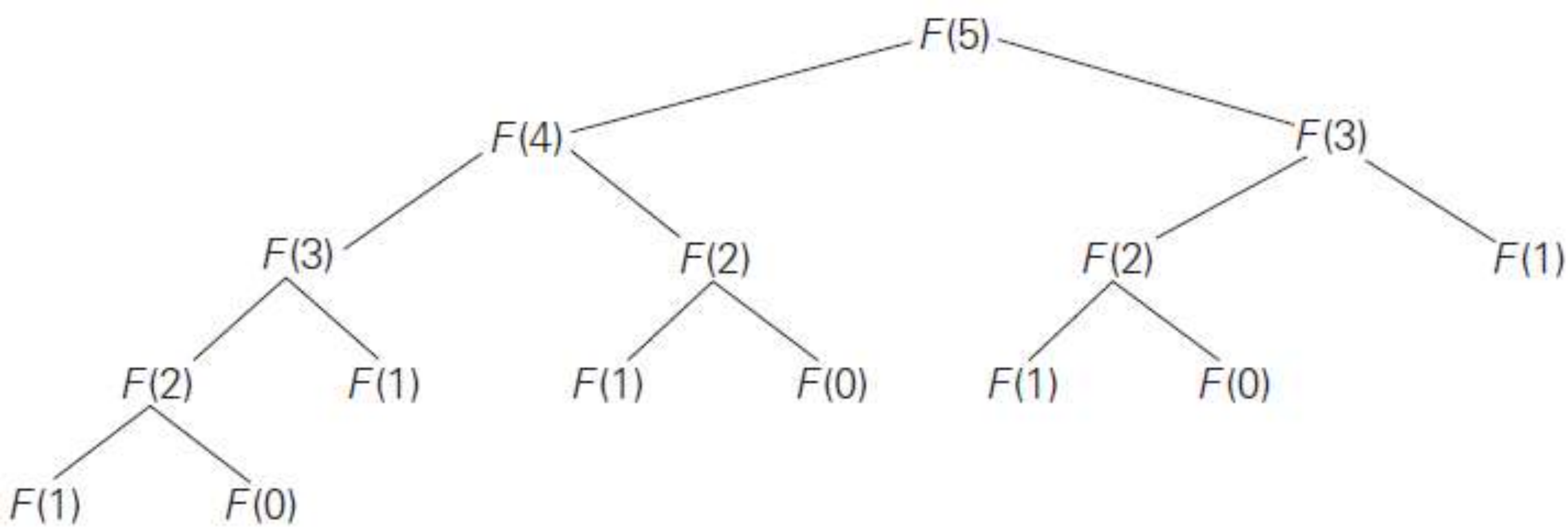
ALGORITHM $Fib(n)$

//Computes the nth Fibonacci number iteratively by using its definition
//Input: A nonnegative integer n
//Output: The nth Fibonacci number
$F[0] \leftarrow 0;\ F[1] \leftarrow 1$
for $i \leftarrow 2$ **to** n **do**
  $F[i] \leftarrow F[i-1] + F[i-2]$
return $F[n]$

$$F(n) = F(n-1) + F(n-2) \quad \text{for } n > 1 \qquad F(0) = 0, \qquad F(1) = 1$$

ALGORITHM $F(n)$

//Computes the nth Fibonacci number recursively by using its definition
//Input: A nonnegative integer n
//Output: The nth Fibonacci number
if $n \le 1$ **return** n
else return $F(n-1) + F(n-2)$



$$A(n) = A(n-1) + A(n-2) + 1 \quad \text{for } n > 1,$$
$$A(0) = 0, \qquad A(1) = 0.$$

$$A(n) - A(n-1) - A(n-2) = 1$$

$$[A(n) + 1] - [A(n-1) + 1] - [A(n-2) + 1] = 0$$

substituting $B(n) = A(n) + 1$:

$$B(n) - B(n-1) - B(n-2) = 0,$$
$$B(0) = 1, \qquad B(1) = 1.$$

$$B(n) = \frac{1}{\sqrt{5}}(\phi^{n+1} - \hat{\phi}^{n+1})$$

where $\phi = (1 + \sqrt{5})/2 \approx 1.61803$ and $\hat{\phi} = -1/\phi \approx -0.61803$

$$A(n) = B(n) - 1 = \frac{1}{\sqrt{5}}(\phi^{n+1} - \hat{\phi}^{n+1}) - 1$$
$$A(n) \in \Theta(\phi^n)$$

`Algorithm: Fib1(n)`

Input Size: **`n`**

Basic Operation : **`Addition`**

C(n) = **1+ C(n - 1) + C(n - 2)**, C(0) = C(1) = 0

$\cong$ $\mathbf{1.6^n} \in \mathbf{\Theta(1.6^n)}$

Note: On a hypothetical computer with 1 terahertz processor and one clock tick per operation, for n = 100, it takes about **8 years** ($\because 1.6^{100} \cong 2^{68}$ operations $\cong 2^3 * 2^{25}$ seconds).

`Algorithm: Fib2(n)`

Input Size: **`n`**

Basic Operation : **`Addition`**

C(n) = **(n - 1)** $\in$ $\mathbf{\Theta(n)}$

Finding nth Fibonacci number:

```
Algorithm Fib3(n)
//
a = 0, b = 1
for i = 2 to n
     c = a + b
     a = b
     b = c
return b
```

```
Algorithm Fib4(n)
//
f = //Closed-form formula
return f
```

$$F_n = \frac{\left(1+\sqrt{5}\right)^n - \left(1-\sqrt{5}\right)^n}{2^n\sqrt{5}}.$$

```
Algorithm Foo(n)
   sum ← 0
   for i ← 1 to n
      j ← 1
      while(j ≤ n)
            sum ← sum + k
            j ← j * 2
      return sum
```

Input Size: n

Basic Operation : **sum ← sum + k**

C(n) **= ?** $\in$ **Θ(?)**

```
Algorithm Foo(n)
   sum ← 0
   i ← 1
   while(i ≤ n)
      for j ← 1 to n
            sum ← sum + k
      i ← i * 2
      return sum
```

Input Size: n

Basic Operation : **sum ← sum + k**

C(n) **= ? ∈ Θ(?)**

```
Algorithm Foo(n)
   sum ← 0
   i ← 1
   while(i ≤ n)
      for j ← 1 to i
            sum ← sum + k
      i ← i * 2
      return sum
```

Input Size: n

Basic Operation : **sum ← sum + k**

C(n) **= ? ∈ Θ(?)**

```
Algorithm Foo(n)
    sum ← 0
    i ← 1
    while(i ≤ n)
        for j ← 1 to i
            for k ← 1 to n in steps of 2
                sum ← sum + k
        i ← i * 2
        return sum
```

Input Size: n

Basic Operation : **sum ← sum + k**

C(n) **= ? ∈ Θ(?)**

Design. Analyse. Repeat!

</ Analysis Framework >