# Analysis of Algorithm

2 algorithms, same task, which is better? Depends!

- e.g. 2 Algorithms to get from Toronto to Waterloo
  1. faster – but need to rent SUV
  2. slower – but scenic & own car

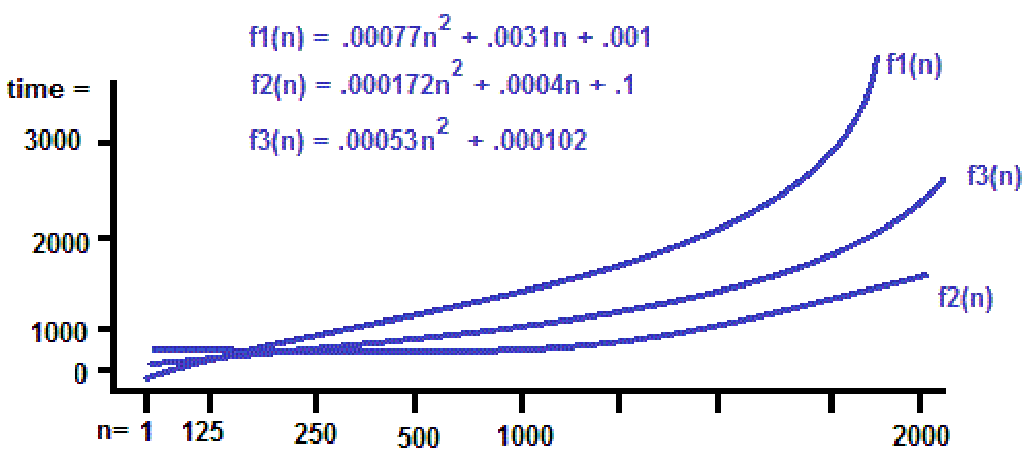
     trading speed vs. cost (vs. scenic)

Computer Algorithms trade-off:

- time
- space (memory used at once)
- disk space
- maintainability

Can't just time them both! Why?

- Consider algorithm A(n)
- Implement A(n) in 3 different environments (programming languages, compilers, Oss, etc.)
- Time each on same/different hardware

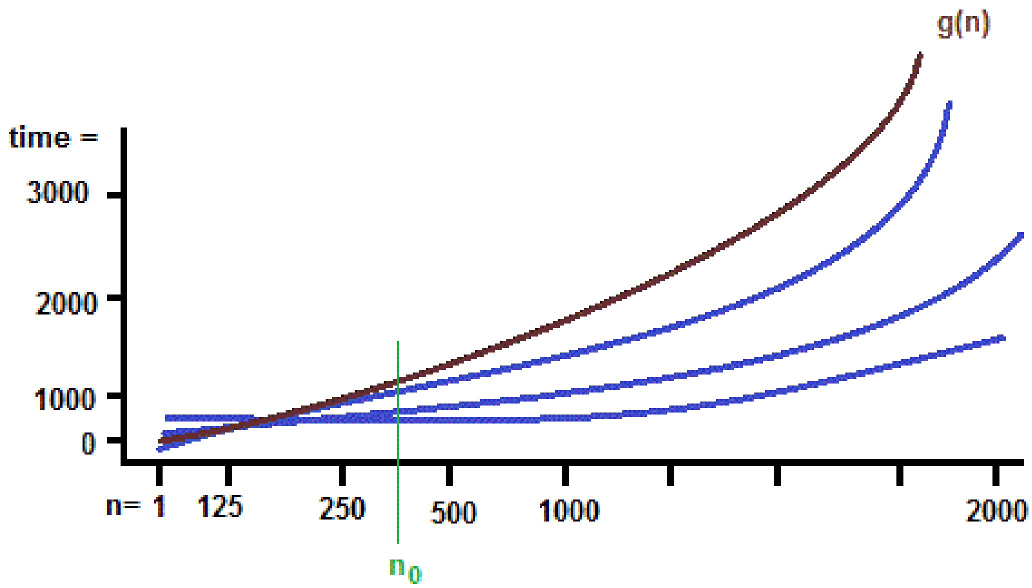


## Big-O Notation

- General measure of efficiency, independent of programming languages, hardware, compiler etc.
- Express algorithm efficiency as a function of problem size ("n")

Formal definition of Big-O Notation

f(n) is O (g(n)) if $\exists$ 2 positive constants, K, $n_0$
such that $|f(n)| \leq K\,|g(n)|$ , $\forall\, n \geq n_0$ (Standish)




e.g. If an algorithm had time complexity
$4n^2 + 3n + 7$ it is $O(n^2)$

- Ignore all but highest powered term
- Ignore all coefficient

$f(n) = 4n^2 + 3n + 7$ $g(n) = n^2$

Proof:

By definition: $4n^2 + 3n + 7 \leq K\,n^2$ $\forall\, n \geq n_0$

$4n^2/n^2 + 3n/n^2 + 7/n^2 \leq K$ pick $n_0 = 1$ (arbitrary chosen because it fits)

$4 + 3 + 7 \leq K$

$14 \leq K$

$4n^2 + 3n + 7 \leq 14\,n^2$ $\forall\, n \geq 1$ TRUE (graph and see)

## Complexity Classes

| | |
|---|---|
| O(1) | Constant |
| O(n) | Linear |
| $O(n^2)$ | Quadratic |
| $O(n^3)$ | Cubic |
| O(logn) | Logarithmic |
| O(nlogn) | Logarithmic |
| $O(2^n)$ | Exponential |
| $O(i^n)$ | Exponential |

N

| N | LogN | NLogN | $N^2$ | $N^3$ | $2^N$ |
|---|---|---|---|---|---|
| 8 | 3 | 24 | 64 | 512 | 256 |
| … | … | … | … | … | … |
| 128 | 7 | 896 | 16, 384 | 2, 097, 152 | 600x > age of universe in nanoseconds |

## Time Complexity

Usually count number of:

- Operations
- Comparisons
- Loop overhead
- Pointer/array refences
- Functional calls (when inside code you're analyzing)

e.g.

```
Sum1 (x) // does 1 + 2 + 3 … + x for positive integer x
        sum = 0
        for (count = 1; count <= x; count++)
                sum = sum+count
        return sum
```

Problem size, n, is the size of integer x
Time?

```
Sum1 (x) //Bdoes 1 + 2 + 3 … + x for positive integer x
C       sum = 0
        for (count D= 1; count <= x; count++)
                sum = sum+Ecount
        return sum
```

Problem size, n, is the size of integer x
Time?

B + nC +nD +nE

$T(n) = B + n(C + D + E)$ or $K_1n + K_0$

➔ O(n) linear

Time taken to sum n integers with Sum 1 is proportional to n

e.g.

```
Sum2 (x) // does 1 + 2 + 3 … + x for positive integer x
        sum = ((x+1) * x)/2
        return sum
```

Problem size, n, is the size of integer x
Time?

```
Sum2 (x) // does 1A + 2B + 3C … +D x for positive integer x
        sum = ((x+1) * x)/2
        return sum
```

Problem size, n, is the size of integer x
Time?

$T(n) = A + B + C + D$ or $K_0$

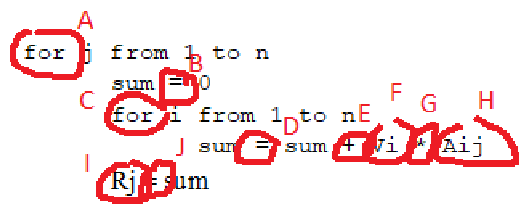
➔ O(1) constant time

e.g.
Algorithm to multiply vector * matrix VA = R
Used in graphics, 3d imagery, medical, flight simulation etc.)

$$[V_1 \quad V_2 \quad \dots \quad V_n]\begin{bmatrix} A_{11} & \cdots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{n1} & \cdots & A_{nn} \end{bmatrix} = [R_1 \quad R_2 \quad \dots \quad R_n]$$

```
for j from 1 to n
      sum = 0
      for i from 1 to n
            sum = sum + Vi * Aij
      Rj = sum
```

Time?




Time?

$T(n) = n(A+B+I+J+ n(C+D+E+F+G+H))$
$= n(K_1 + n(K_2))$
$= K_2n^2 + K_1n$
➔ $O(n^2)$ Quadratic

e.g.
Find O of G for an algorithm that searches for items in an array (count comparisons only
**Comparisons are only counted if it's one item with another, not empty() function**

```
Find (A, I)
      if empty(A) return NO
      for i from 1 to n
            if A[i] == I return YES
      return NO
```

For finding the item in the n-item array:

- Best Case: I at A[0] 1 comp -> O(1)
- Worst Case: I at A[n-1] n comps -> O(n)
- Average Case: Average of finding item at A[0], A[1] .. A[n-1]

  At A[1] 1 comp
  At A[2] 2 comps
  …
  At A[n-1] n comps

  Average? 1+2+3+…+n / n
  = n(n+1)/2n = 1/2n + ½

  ➔ O(n)

Recursive version

```
Find (A, I)
      if empty(A) return NO
      if I == first (a)return YES
      return find (AllButFirst(A), I)
```

Number of comps for Find when A has n items

Base Case: T(1) = 1
Recurrence Relation: T(n) = 1 + T(n-1)

T(n) = 1 + T(n-1)
= 1 + 1 + T(n-2) = 2 + T(n-2)
= 1+ 1 + 1 + T(n-3) = 3 + T(n-3)
= 1+ 1 + 1 + 1 + T(n-4) = 4 + T(n-4)
… (unroll)
= (n-1) + T(n-(n-1))
= n -1 +T(1)
= n – 1 + 1 = n

T(n) = n is O(n)

- Best Case: I at A[0] 1 comp -> O(1)
- Worst Case: I at A[n-1] n comps -> O(n)

- Average Case: Average of finding item at A[0], A[1] .. A[n]

  Remember that T(n) = n, therefore

  At A[0] 1 ~~comp~~
  At A[1] 2 comps
  …
  At A[n-1] n comps

  Average? 1+2+3+…+n / n
  = n(n+1)/2n = (½)n + ½

  ➔ O(n)

## Space Complexity

Measure amount of memory used AT ONCE by algorithms

- Instruction space (memory to hold compiled version of program constant for any n)
- Data space (variables, data structures, allocated memory)
- Environment Space (constant for each function call)

e.g.
Iterative factorial:

```
PosFact1(x)
      prod = 1
      for i  from 2 to x
            prod = prod * i
      return prod
```

Count: function, x, prod, i = K
S(n) = k is O(1)

Recursive factorial:

```
PosFact2(x)
      if x <= 1 return 1
      return (PosFact2(x-1) * x)
```

Count: function, x, prod = k + memory for PosFact(n-1)
Base case: S(1) = k
Recurrence Relations: S(n) = k + S(n-1)

$$
\begin{aligned}
S(n) &= k + S(n-1) \\
&= k + k + S(n-2) \qquad = 2k + S(n-2) \\
&= k + k + k + S(n-3) \qquad = 3k + S(n-3) \\
&\dots \text{(unroll)} \\
&= (n-1)k + S(n-(n-1)) \\
&= (n-1)k + S(1) \\
&= (n-1)k + k = nk \\
&\Rightarrow O(n)
\end{aligned}
$$

e.g.

```
fib(x)              // x >= 0
        if x <= 1   return 1
        return fib(x-1) + fib(x-2)
```

Space Complexity?

```
                      n
                /           \
             n-1             n-2
           /      \        /      \
        n-2     n-3     n-3      n-4
       /    \     /   \
    n-3    n-4  n-4   n-5    …        …
      /      …    …     …
    1   0  …
```

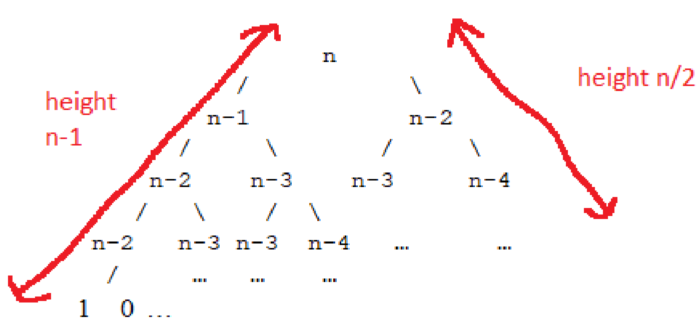
For each call (node) space is: fib + x = k have n nodes in longest path,
Therefore, max space used AT ONCE is nk is O(n)

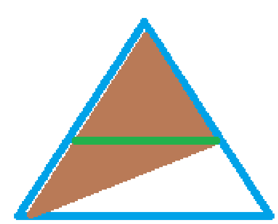
Time complexity for fib?

```
fib(x)              // x >= 0
        if x <= 1   return 1
        return fib(x-1) + fib(x-2)
```

If it's too difficult to unroll recurrence relation
For time, count every call:

- Each call takes {k1 internal node, k0 leaf } say k
- How many calls? Equals to number of nodes in fib tree



| | | | | |
|---|---|---|---|---|
| $2^{(n/2+1)}$-1 | < | # nodes | > | $2^n$ -1 |
| $2*2^{(n/2)}$-1 | < | # nodes | > | $2^n$ -1 |
| $2*2^{(1/2)}$-1 | < | # nodes | > | $2^n$ -1 |

Time is between $k*2*2^{(n/2)} - k$ & $k2^n - k$

➔ O ($2^n$) exponential in time

Fib Lower Bound:

Base case: T(0)=T(1)= 1
Recurrence Relation: T(n) = T(n-1) + T(n-2) + k

$T(n) = T(n-1) + T(n-2) + k$ T(n-1) >= T(n-2) ? YES and so
$\geq T(n-2) + T(n-2) + k$
$\geq 2T(n-2) + k$
$\geq 2(T(n-3) + T(n-4) + k) + k$
$\geq 2(T(n-4) + T(n-4) + k) + k$
$\geq 2^2T(n-4) + 2^1k + k$ $= 2^2T(n-4) + 3k$
$\geq 2^2(2T(n-6) + k) + k$
$\geq 2^3(T(n-6) + 2^2k) + k$ $= 2^3T(n-6) + 7k$
…
$\geq 2^x(T(n-2x) + (2^x - 1)k$
let $n-2x = 0$ from making T(n-2x) eventually equals T(0)
$x = n/2$
substitute n/2 for x in

$T(n) \geq 2^{n/2}T(0) + (2^{n/2} - 1)k \geq 2^{n/2} + 2^{n/2}k - k$
➔ $O(2^{n/2})$

Fib Upper Bound:

Base case: T(0)=T(1)= 1
Recurrence Relation: T(n) = T(n-1) + T(n-2) + k

$T(n) = T(n-1) + T(n-2) + k$ T(n-1) >= T(n-2) ? YES and so
$\leq T(n-1) + T(n-1) + k$
$\leq 2T(n-1) + k$
$\leq 2(T(n-2) + T(n-3) + k) + k$
$\leq 2(T(n-2) + T(n-2) + k) + k$
$\leq 2^2T(n-2) + 2^1k + k$ $= 2^2T(n-3) + 3k$
$\leq 2^2(2T(n-3) + k) + k$
$\leq 2^3(T(n-3) + 2^2k) + k$ $= 2^3T(n-4) + 7k$
…
$\leq 2^x(T(n-x) + (2^x - 1)k$
let $n-x = 0$ from making T(n-x) eventually equals T(0)
$x = n$
substitute n for x in

$T(n) \geq 2^nT(0) + 2^nk - k \geq 2^n + 2^nk - k$
➔ $O(2^n)$
$2^{n/2} \leq T(n) \geq 2^n$
T(n) is $O(2^n)$

Towers of Hanoi

```
Move (n, from, to, temp)
      if (n > 0)
            move (n-1, from, temp, to)
            take nth     off from , Put on to
            move (n-1, temp, to, from)

                              n
                      /                  \
                   n-1                     n-1
              /           \            /          \
            n-2          n-2          n-2          n-2
            …                                         …
         0                    …                          0
```

Space?

Height of a binary tree is the longest path from root to leaf, which in this case is n +1
Space for 1 call is k (move, n, from, to, temp) # calls in memory at once?
S(n) = kn + k is O(n)

Time?
For 1 call time is k(for ops: >, -,+,take, put)
Total number calls?
Total number of nodes ($2^{(n+1)} - 1$), Therefore $T(n) = k(2^{(n+1)} - 1)$ is $O(2^n)$

Base case: T(0)=k
Recurrence Relation: T(n) = k + T(n-1) + T(n-1)

$$
\begin{aligned}
T(n) &= k + T(n-1) + T(n-1) \\
&= k + 2T(n-1) \\
&= k + 2(k + T(n-2) + T(n-2)) \\
&= k + 2(k + 2T(n-2)) \\
&= k + 2(k + 2(k + T(n-3) + T(n-3))) \\
&= k + 2(k + 2(k + 2T(n-3))) \\
&= k + 2(k + 2(k + 2(k + 2T(n-4)))) \\
&= 2^4T(n-4) + 2^3k + 2^2k + 2^1k + 2^0k \\
&\dots \\
&= 2^nT(0) + 2^{n-1}k + 2^{n-2}k + 2^{n-3}k + \dots + 2^0k \\
&= 2^nk + 2^{n-1}k + 2^{n-2}k + 2^{n-3}k + \dots + 2^0k \\
&= k\sum_{i=0}^{n} 2 \quad = \quad k(2^{n+1} - 1)
\end{aligned}
$$

➔ $O(2^n)$

Binary Search
Given ordered array A, key K, return K's index in A

e.g.

A = {2,4,9,12,15,21,25,29,30,32,36,37,38,39,42}
BinSearch (A, 9, 0, 14) -> 2
BinSearch (A, 42, 0, 14) -> 14

```
BinSearch (A, K, L, R)
      if (L > R)        return -1                //not found
      M = midpointBetween (L,R)
      if (K == A[M])          return M                 //found
      if (K > A[M]) return    BinSearch (A, K, M+1, R) //Check right half
      else              BinSearch (A, K, L, M-1)      //Check left half
```

Worst case time for finding item?

Recursively call until L = R
T(n) = worst case number comps for finding k in A of n items
Base case: $T(1) = 1$
Recurrence Relation: $T(n) = 2 + T(n/2)$

$$
\begin{aligned}
T(n) &= 2 + T(n/2) && = \\
&= 2 + 2 + T(n/4) && = 2*2 + T(n/2^2) \\
&= 2 + 2 + 2 + T(n/8) && = 2*3 + T(n/2^3) \\
&= 2 + 2 + 2 + 2 + T(n/16) && = 2*4 + T(n/2^4) \\
&\dots \\
&= 2*k + T(n/2^k)
\end{aligned}
$$

$1 = n/2^k$ from base case
$k\log_2 2 = \log_2 n$ from $\log a^b = b\log a$
$k = \log_2 n$

substitute k back in
$T(n) = 2*\log_2 n + T(n/2\log_2 n)$
$= 2*\log_2 n + T(n/n) = 2*\log_2 n + 1$

➔ O (logn) Logarithmic time for worst case comps for finding K

Another way to look at Binary Search

| Size of List (# of elements) | Time to find particular key(how many cmps to look at T(n)) |
|---|---|
| 1 | 1 |
| 3 | 2 |
| 7 | 3 |
| 15 | 4 |
| 31 | 5 |
| … | … |
| $2^k - 1$ | k |

$n = ?$
$= 2^k - 1$
$n + 1 = 2^k$
$\log_2(n+1) = \log_2(2^k)$
$\log_2(n+1) = k\log_2(2)$
$\log_2(n+1) = k$

$T(n) = \log_2(n+1)$
➔ O (logn) Logarithmic time for worst case comps for finding K