# Introduction to Efficiency

# What Makes a Good Program

- It works as specified.
- It is easy to understand and modify.
  - *Decomposition*
    - Stepwise refinement: Create structure by decomposing program into meaningful units called *modules* (*functions* and *classes*).
  - *Abstraction*
    - Separate the “contractual” interface of an abstraction from its implementation (*information hiding*).
  - *Generalization*
    - Design functions that have a good chance of being reused elsewhere.
- It is reasonably efficient: time and memory.

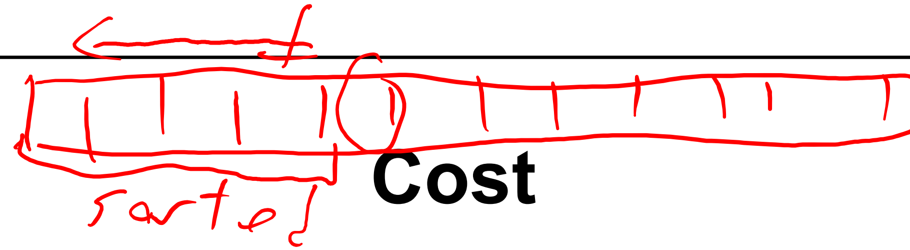
# Goals

- Describe a formal yet practical model of efficiency.
- Understand why some programs take much longer than others.
- Learn to predict the running time of your programs.
- Understand that *efficiency is about algorithms,* not about specific language features or fast computers.
- Learn various techniques that allow you to design programs that will finish by a target deadline.
- Learn about *reduction techniques.*
  - Convert a given problem into a different problem that can be solved efficiently.
- Understand the trade-offs between time and memory.

# Algorithm Efficiency

- A first attempt: *An algorithm is efficient if, when implemented, it runs quickly on real input instances.*
- What is missing?
  - What does “quickly” really mean?
  - Run where?
  - Run on what inputs?
  - Implemented how and in what language?
  - How do you compare with other algorithms?
  - How does the performance scale up?

# Example: Insertion Sort




| | `def insertionSort(A):` | **Cost** |
|---|---|---|
| 1. | `for j in range(1,len(A)):` | $c_1 n$ |
| 2. | `    k = A[j]` | $c_2(n-1)$ |
| 3. | `    i = j–1` | $c_3(n-1)$ |
| 4. | `    while i>=0 and A[i]>k:` | $c_4 \sum_{j=2}^{n} t_j$ |
| 5. | `        A[i+1]=A[i]` | $c_5 \sum_{j=2}^{n} (t_j - 1)$ |
| 6. | `        i = i–1` | $c_6 \sum_{j=2}^{n} (t_j - 1)$ |
| 7. | `    A[i+1]=k` | $c_7(n-1)$ |

$$T(n) = an + b\left(\sum_{j=2}^{n} t_j\right) + c$$

# Algorithm Efficiency

- The exact running time depends on the constants $c_1, c_2, \ldots$ involved in the previous calculation, the input size $(n)$, and the actual input.
- These constants, in turn, depend on the computer and language used and are hard to compute exactly without implementing the algorithm and taking exact measurements.
- You need a definition that is independent of platform and language and has predictive value as the problem scales up.

END

# Best, Worst,
and Average Efficiency

# Algorithm Efficiency

- A second attempt: *An algorithm is efficient if, when implemented on any computer and language, it performs a small number of basic steps on real input instances.*
- A basic step is a block of instructions that takes a constant amount of time each time it is executed (i.e., the time required for one execution is independent of the input size).
- An algorithm *A* is better than algorithm *B* if *A* performs fewer basic steps than *B*.

# Example Revisited

| | def insertionSort(A): | **Cost** |
|---|---|---|
| 1. | for j in range(1,len(A)): | $n$ |
| 2. | k = A[j] | $n-1$ |
| 3. | i = j–1 | $n-1$ |
| 4. | while i>=0 and A[i]>k: | $\sum_{j=2}^{n} t_j$ |
| 5. | A[i+1]=A[i] | $\sum_{j=2}^{n} (t_j-1)$ |
| 6. | i = i–1 | $\sum_{j=2}^{n} (t_j-1)$ |
| 7. | A[i+1]=k | $n-1$ |

$$T(n) = an + b\left(\sum_{j=2}^{n} t_j\right) + c$$

# Algorithm Efficiency

- To understand the performance of algorithm *A*, it is not enough to run it on one input.
- You need to understand behavior (memory, running time) over *all* possible input instances.
  - Minimum
  - Maximum
  - Average

  over all inputs of size *n*
- Computational complexity (cost) is usually expressed as a function (e.g., a polynomial) of the input size *n.*

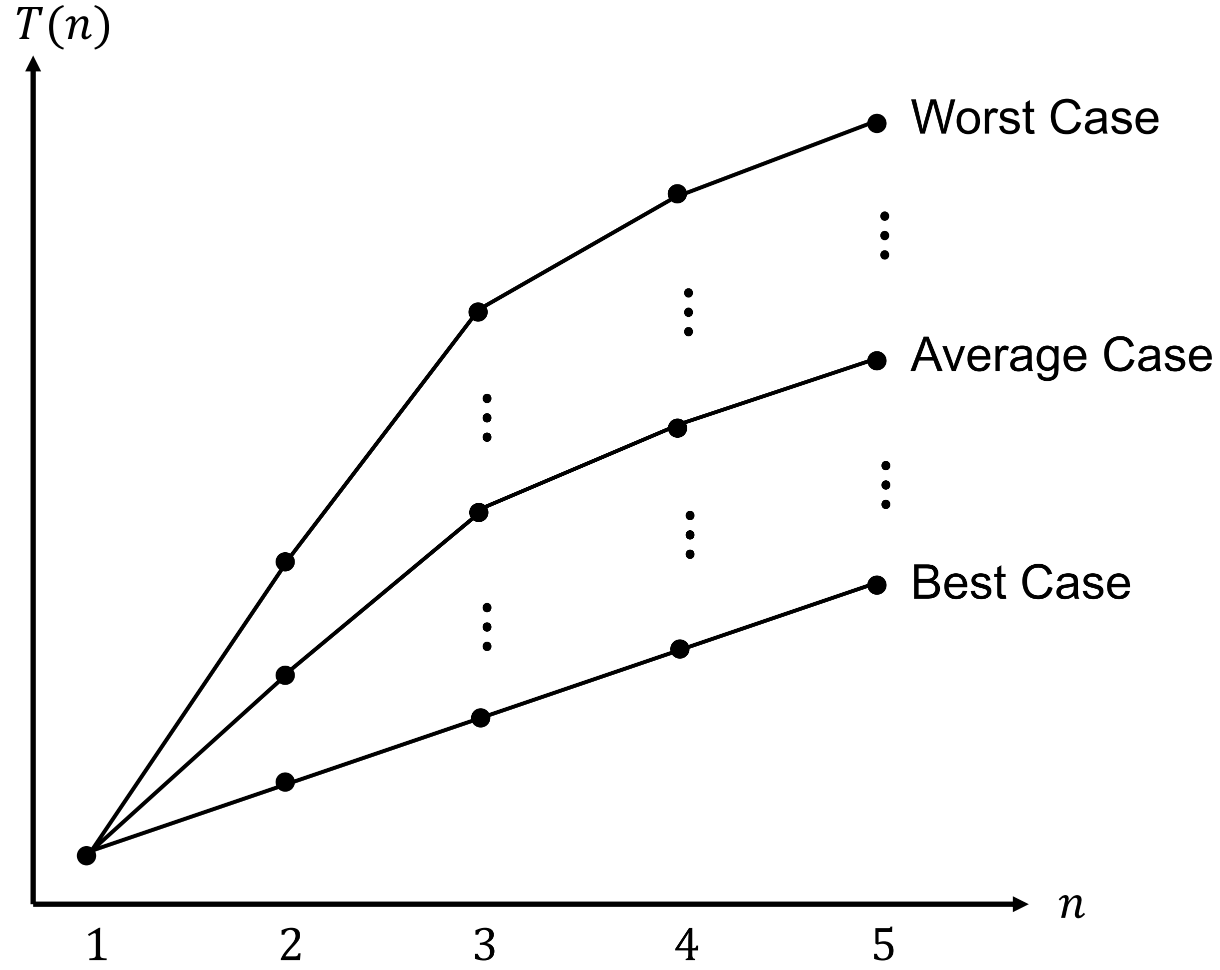
*Question:* What input distribution should be used when computing the average?

# Worst, Best, and Average

- The *worst-case* complexity is the function defined by the *maximum* time taken on any instance of size *n*.
- The *best-case* complexity is the function defined by the *minimum* time taken on any instance of size *n*.
- The *average-case* complexity is the function defined by an *average* time taken on any instance of size *n*.
- **Each of these is a function $\mathbf{T}: \mathbb{N} \to \mathbb{R}^{+}$ that maps input size to cost (e.g., time, # of steps, etc.).**

# Performance of a Sorting Algorithm $A$

For every instance $I$, run $A$ and plot point ($|I|$, $T_A(|I|)$ ).



# Insertion Sort: Analysis

Worst, best, and average depend on the values $t_j$.

- Best case: $t_j = 1 \Rightarrow$

  $$T(n) = an + b(n - 1) + c = A_1 n + A_0$$

- Worst case: $t_j = j \Rightarrow$

  $$T(n) = an + b(n + 2)(n - 1)/2 + c = B_2 n^2 + B_1 n + B_0$$

- Average case: $t_j = j/2$

  $$T(n) = an + b(n + 2)(n - 1)/4 + c = C_2 n^2 + C_1 n + C_0$$

$$1 + 2 + 3 + 4 + 5 \ldots + n = \frac{n(n-1)}{2}$$

$$\frac{n^2 - n}{2}$$

# Exact Analysis Is Difficult

- As before, best, worst, and average case are difficult to deal with precisely because too many details
- Exact values of constants $(a_i, b_i, d_i, \ldots)$ depend on machine/language used, how you implemented the algorithm, and so on.

$\Rightarrow$ "Exact" analysis is not general.

- Remember, count the number of *basic steps* instead.
  - A basic step is an operation that takes constant time.
  - $T: \mathbb{N} \rightarrow \mathbb{N}$ **that maps input size to total # basic steps**
- Third attempt: *An algorithm is efficient if, for large inputs, it performs* ***significantly*** *fewer basic steps than a naïve or brute force approach.*

# Insertion Sort: Analysis

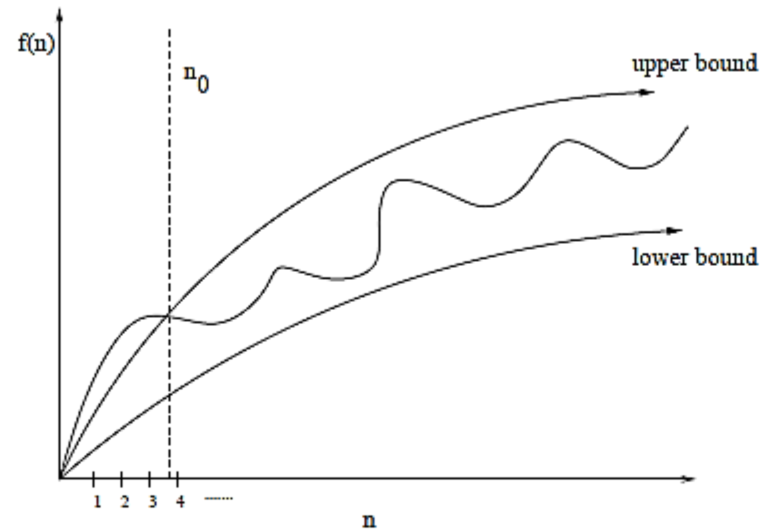
- Best case: $\boldsymbol{T(n) = a_1 n + a_0}$
- Worst case: $\boldsymbol{T(n) = b_2 n^2 + b_1 n + b_0}$
- Average case: $\boldsymbol{T(n) = c_2 n^2 + c_1 n + c_0}$
- How much do the lower-order terms contribute to $T(n)$?
  - Insignificant for large $n$

END

# Efficiency Bounds

# A Simpler Approach

- It is easier to talk about upper and lower bounds of the function in a manner that avoids machine and implementation details.
  - Ignore machine-dependent constants.
  - Drop lower-order terms.
  - Look at growth rate as *T*(*n*).

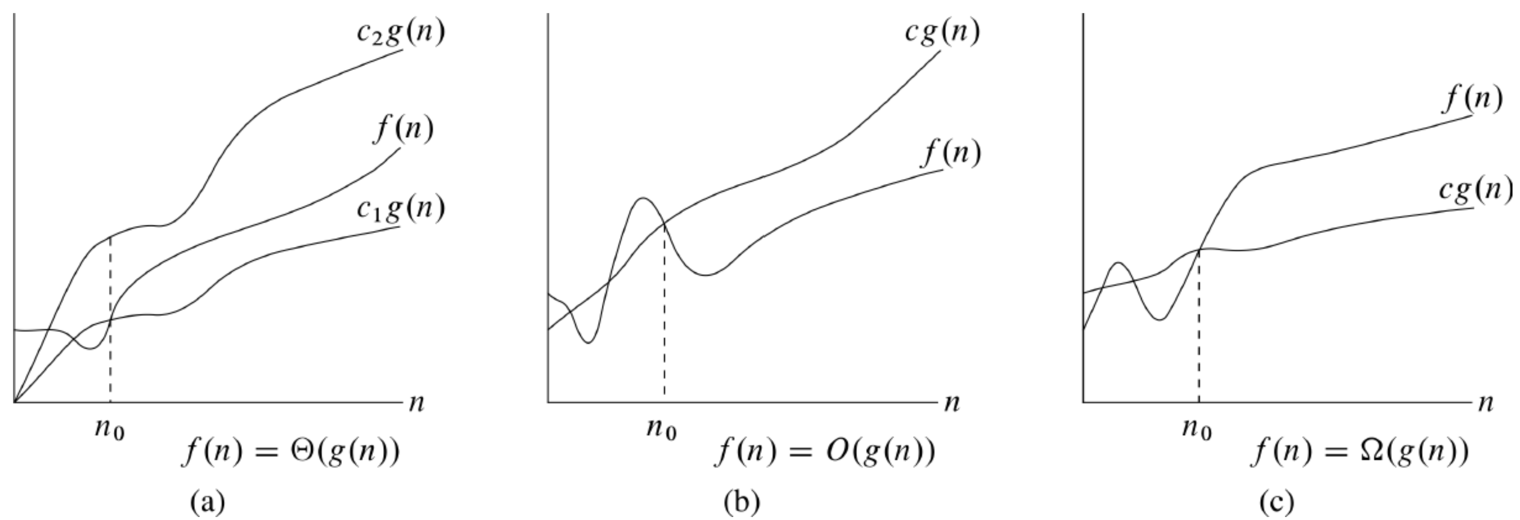


# Complexity Classes

- $f(n) \in O(g(n))$ means that $c \cdot g(n)$ is upper bound for $f(n)$.
- $f(n) \in \Omega(g(n))$ means that $c \cdot g(n)$ is lower bound for $f(n)$.
- $f(n) \in \Theta(g(n))$ means that $c_1 \cdot g(n)$ is upper bound for $f(n)$ and $c_2 \cdot g(n)$ is lower bound for $f(n)$.

$c$, $c_1$, and $c_2$ are constants independent of $n$; bound holds for "sufficiently large" $n$.

CS people (unfortunately) tend to use $=$ instead of $\in$.



# Example: Insertion Sort

$$\text{Time}: \quad T(n) = a_1 n + a_2 \cdot \sum_{j=2}^{n} t_j + a_3$$

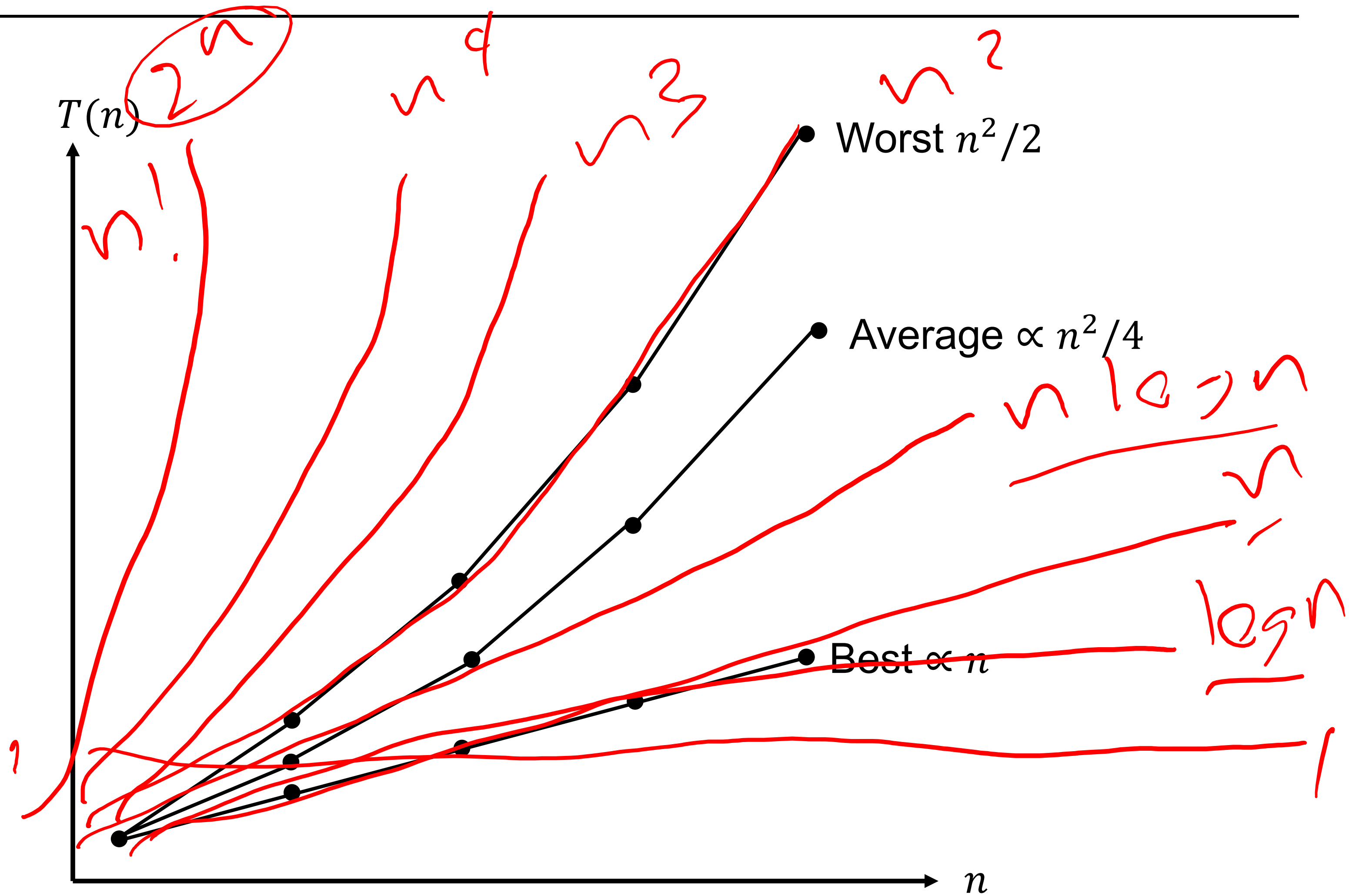
$$(a_1 + a_2)n + a_3 - 1 \le T(n) \le a_1 n + a_2 \cdot \frac{(n+2)(n-1)}{2} + a_3$$

- Worst case grows as $n^2 \Rightarrow T_{max}(n) \in \Theta(n^2)$
- Best case grows as $n \Rightarrow T_{min}(n) \in \Theta(n)$
- Average case grows as $n^2 \Rightarrow T_{ave}(n) \in \Theta(n^2)$

$\Rightarrow$ insertion sort takes "between" $c_1$n and $c_2 n^2$

That is, $T(n) \in \Omega(n)$ and $T(n) \in O(n^2)$

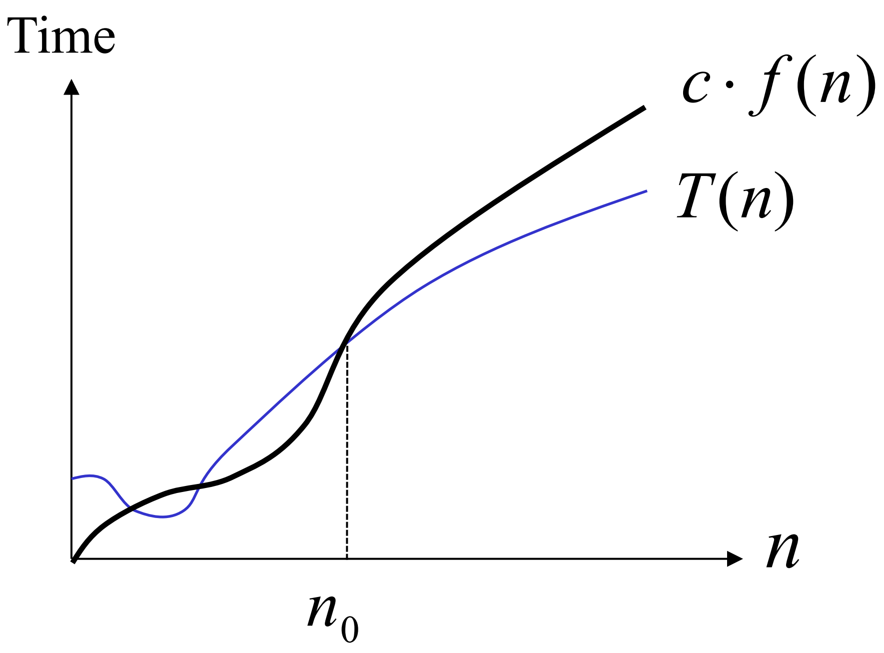
# Performance of Insertion Sort



# Asymptotic Notation: O

Suggests the idea of upper bound for *T*(*n*)

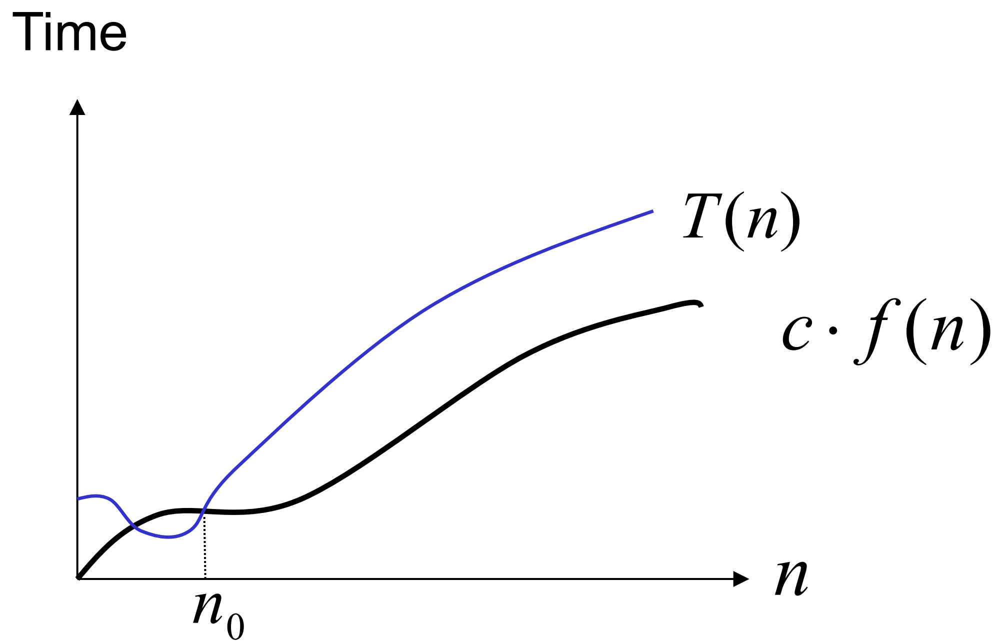
$$T(n) \in O(f(n)) \Leftrightarrow \exists c, n_0 : T(n) \le c \cdot f(n) \ \forall n \ge n_0$$



# Asymptotic Notation: $\Omega$

Suggests the idea of lower bound for *T(n)*

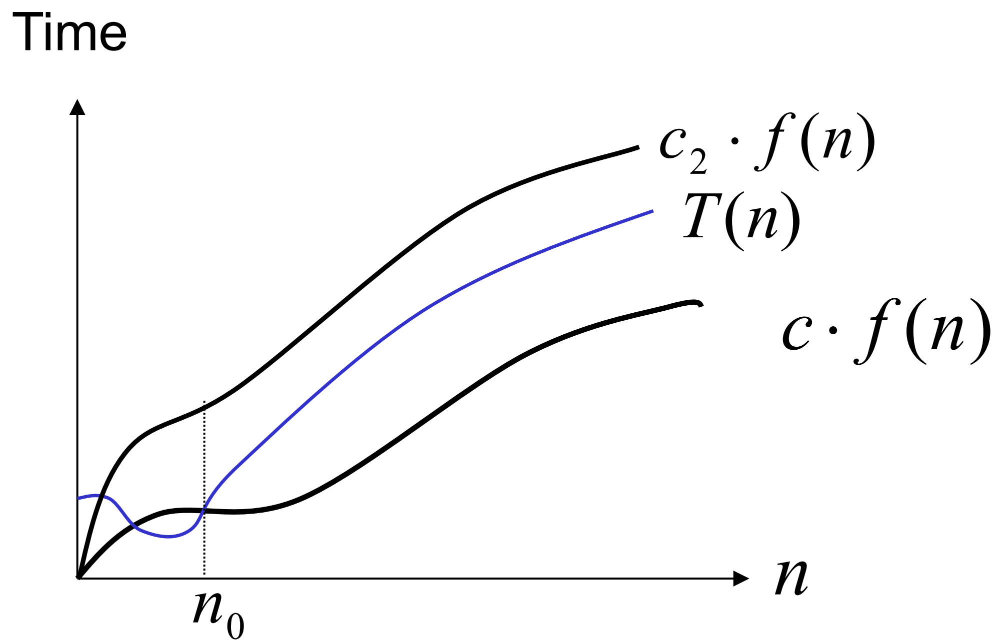
$$T(n) \in O(f(n)) \Leftrightarrow \exists c, n_0 : T(n) \leq c \cdot f(n)\ \forall n \geq n_0$$



# Asymptotic Notation: $\Theta$

Suggests an *equivalent bound* for *T*(*n*)

$$T(n) \in \Theta\big(f(n)\big) \iff T(n) \in O\big(f(n)\big) \text{ and } T(n) \in \Omega(f(n))$$



# Examples: O, Ω, Θ

$3n^2 - 50n + 5 \in O(n^2)$ because $3n^2 > 3n^2 - 50n + 5$
$3n^2 - 50n + 5 \in O(n^3)$ because $0.1n^3 > 3n^2 - 50n + 5$
$3n^2 - 50n + 5 \notin O(n)$ because $cn < 3n^2$ whenever $n > c$

$3n^2 - 50n + 5 \in \Omega(n^2)$ because $2n^2 < 3n^2 - 50n + 5$
$3n^2 - 50n + 5 \in \Omega(n)$ because $2n < 3n^2 - 50n + 5$
$3n^2 - 50n + 5 \notin \Omega(n^3)$ because $3n^2 - 50n + 5 < n^3$

$O(1)$

$3n^2 - 50n + 5 \in \Theta(n^2)$ because both $O$ and $\Omega$
$3n^2 - 50n + 5 \notin \Theta(n^3)$ because $O$ only
$3n^2 - 50n + 5 \notin \Theta(n)$ because $\Omega$ only

$O(2^n)$

# Addition and Multiplication of Functions

Suppose that $f(n) \in O(n^2)$, $f(n) \in \Omega(n)$, and $g(n) \in \Theta(n^2)$.

- What do we know about f(n) + g(n)?

  $\boldsymbol{f(n) + g(n) \in O(n^2),\ f(n) + g(n) \in \Omega(n^2)}$

- How about c·f(n)?

  $\boldsymbol{c \cdot f(n) \in O(n^2),\ c \cdot f(n) \in \Omega(n)}$

- How about f(n) · g(n)?

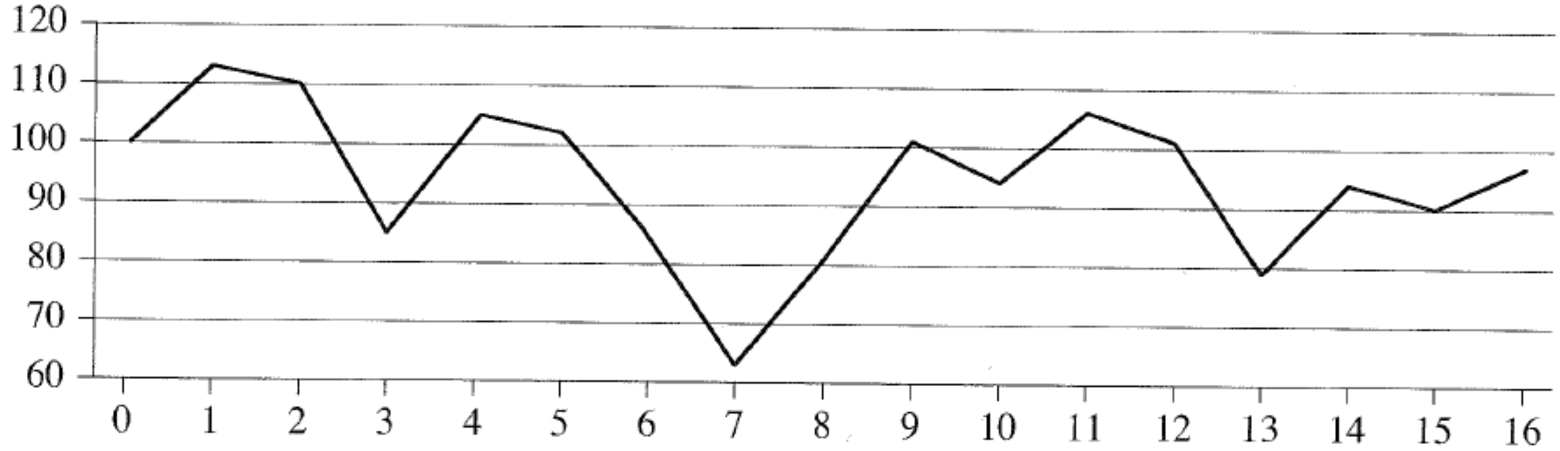
  $\boldsymbol{f(n) \cdot g(n) \in O(n^4),\ f(n) \cdot g(n) \in \Omega(n^3)}$

END

# Stock Profit

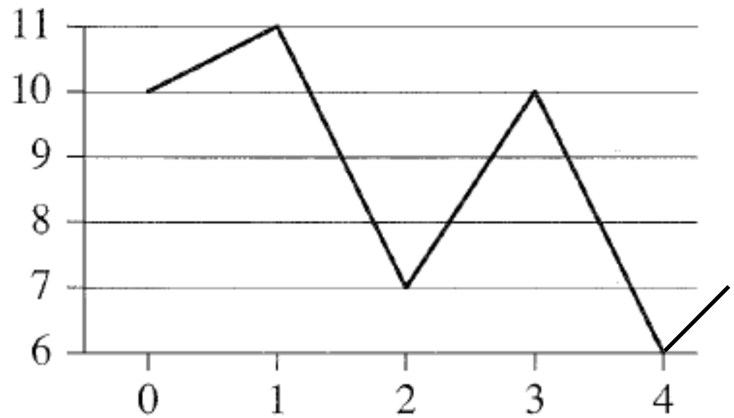
# Example:
# Maximum Stock Trading Profit

When was the best time to buy and sell in order to maximize profit?



# Proposed Algorithms

- **Algorithm 1:** Buy at the lowest point and sell at the highest point after it.
- **Algorithm 2:** Sell at the highest point and buy at the lowest point before it.
- **Algorithm 3:** Choose the best of **1** or **2.**




- **Algorithm 4:** Consider all pairs (*i*,*j*) of days where $j > i$ and choose the best pair.

# Maximizing Stock Profit

```python
def MaxProfit(A):
    """Given a list A of stock prices, find the buy and
sell
        times that maximize profit"""
    best,buy,sell = 0,0,-1
    n = len(A)
    for i in range(n):
        for j in range(i,n):
            profit = A[j]-A[i]
            if profit > best:
                best,buy,sell = profit,i,j
    return best,buy,sell
```

$n$

$n-1$

$n-2$

$\vdots$

$+ \ 1$

What is the complexity class of MaxProfit ? $T(n) \in \Theta(n^2)$

$\frac{n(n-1)}{2}$

END

# Matrix Multiplication

# Matrix Multiplication

```
def mulMatrix(A,B):
    """returns the product of 2 matrices"""
    rowA = len(A)
    colA = len(A[0])
    rowB = len(B)
    colB = len(B[0])
    assert colA == rowB
    C = newMatrix(rowA,colB)
    …

    for i in range(rowA):
        for j in range(colB):
            sum = 0
            for k in range(colA):
                sum += A[i][k]*B[k][j]
            C[i][j] = sum
    return C
```

What is the complexity $T(n)$ when multiplying two $n \times n$ matrices?

$$T(n) \in \Theta(n\sqrt{n})$$

If $d$ is the dimension of a $d \times d$ matrix, then $T(d) \in \Theta(d^3)$.

$d^3$

END

# Search

# Searching a List

- Let *L* be a list with elements drawn from a domain *D*.
- A fundamental operation on *L* is that of searching for an arbitrary member $x \in D$.
- Solutions to this problem depend on whether the elements of *L* appear in order.
  - *Linear search*: Examine elements in the given order until *x* is found or the list is exhausted.
  - *Binary search*: If the median element is not *x*, proceed with one-half of the list only.

# Linear Search of a Sorted List

```
def linSearch(L, target):
    for e in L:
        if e == target:
            return True
        if e > target:
            return False
    return False
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 3 | 4 | 7 | 8 | 11 | 15 | 20 |

n/2

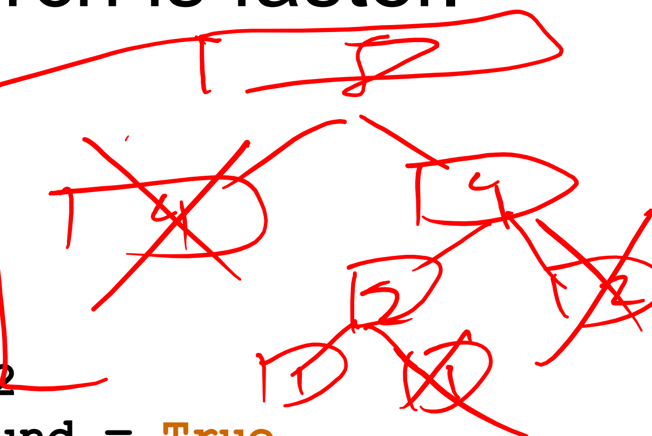
n

Complexity? $T(n) \in O(n)$

Worst case: $\Theta(n)$ Best case: $\Theta(1)$

# Binary Search of a Sorted List

If the list is sorted, binary search is faster.



```
def binSearch(L, item):
    low = 0
    high = len(L)-1
    found = False
    while low <= high and not found:
        midpoint = (first + last)//2
        if L[midpoint] == item:   found = True
        elif item < L[midpoint]:  last = midpoint-1
        else: first = midpoint+1
    return found
```

$\log_2 8 = 3$

$2^3 = 8$

Complexity? $T(n) \in O(\log n)$

Worst case: $\Theta(\log n)$ Best case: $\Theta(1)$

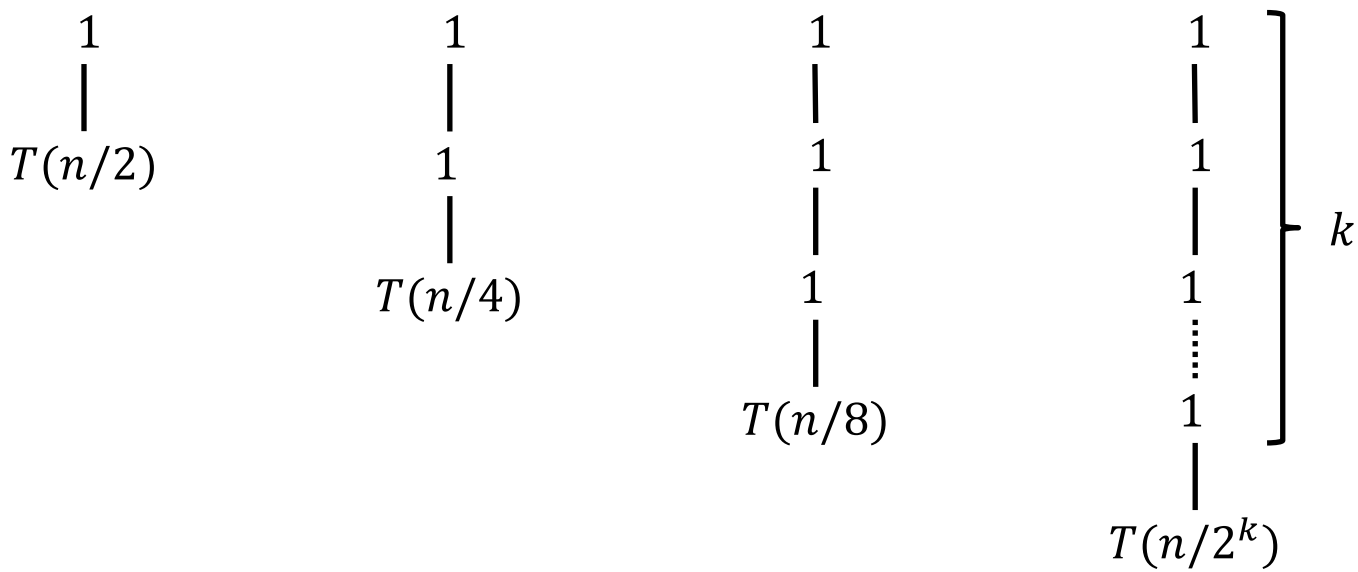
# Binary Search Revisited

```
def binsearch(L, target, low, high):
     if  high-low < 2:
           return L[low]==target or L[high]==target
     mid = (low + high) // 2
     if L[mid] == target:
           return True
     if L[mid] > target:
           return binsearch(L, target, low, mid – 1)
     else:
           return binsearch(L, target, mid+1, high)
```

Complexity can also be described recursively.

$$T(n) = 1 + T(n/2) \in \Theta(\log n)$$

# Analysis




$$T(n) = 1 + T(n/2) = 2 + T(n/4) = \cdots = k + T\left(n/2^k\right) \in \Theta(\log n)$$

**END**

# Acknowledgements


- Introduction to Algorithms, 3rd edition, by T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein; MIT Press, 2009
- Algorithm Design Manual 2ed, Steven Skiena, Springer.2010.