# Algorithms Week 1



Ljubomir Perković, DePaul University

We will cover techniques for designing computer algorithms, including:

- divide-and-conquer
- backtracking
- dynamic programming
- greedy

We will later use graph algorithms as a case study.

Throughout, we will also make use of techniques for analyzing computer algorithms and problems including, towards the end of the course, the theory of NP-completeness.

## Course mechanics

See the course web site https://reed.cs.depaul.edu/lperkovic/courses/csc421

- Find there the course syllabus
- My weekly lecture slides and video recordings will be posted there on Mondays



I will hold optional, recorded Zoom discussion sessions to discuss the week's topics on Wednesdays



- The discussion session recording will be posted on the course web site
- The weekly homework assignment will be posted on Wednesday as well
- The homework assignment will be due the following Wednesday



A final exam will be given at the end of the course.

# Contact



Discussion sessions: Wed, 7:30pm-9:00pm CT, Zoom meeting link in D2L

Office hours: Mon, 3:00pm-4:30pm and 7:30pm-9:00pm, Zoom meeting link in D2L



Discussion forum: Discord server link in D2L



E-mail: lperkovic@cs.depaul.edu

Phone: 312-362-8337 (leave a message)

An algorithm is a step-by-step procedure for solving a problem

- Typically developed before doing any programming
- In fact, it is independent of any programming language



Efficient algorithms can have a dramatic effect on our problem-solving capabilities

The issues that will concern us when developing algorithms:

The issues that will concern us when developing algorithms:

1. Problem specification - Is the problem clearly and precisely stated?

The issues that will concern us when developing algorithms:

1. Problem specification - Is the problem clearly and precisely stated?
2. Simplicity and clarity - Is the algorithm clear? Is there a simpler and clearer algorithm?

The issues that will concern us when developing algorithms:

1. Problem specification - Is the problem clearly and precisely stated?
2. Simplicity and clarity - Is the algorithm clear? Is there a simpler and clearer algorithm?
3. Algorithm correctness - Is the algorithm correct?

The issues that will concern us when developing algorithms:

1. Problem specification - Is the problem clearly and precisely stated?
2. Simplicity and clarity - Is the algorithm clear? Is there a simpler and clearer algorithm?
3. Algorithm correctness - Is the algorithm correct?
4. Amount of work done - What is the running time of the algorithm in terms of the input size (independent of hardware and programming language)?

The issues that will concern us when developing algorithms:

1. Problem specification - Is the problem clearly and precisely stated?
2. Simplicity and clarity - Is the algorithm clear? Is there a simpler and clearer algorithm?
3. Algorithm correctness - Is the algorithm correct?
4. Amount of work done - What is the running time of the algorithm in terms of the input size (independent of hardware and programming language)?
5. (Sometimes) amount of space used - How much extra memory space does the algorithm use (here we mean the amount of extra space beyond the size of the input). We will say that an algorithm is *in place* if the amount of extra space is constant with respect to input size.

The issues that will concern us when developing algorithms:

1. Problem specification - Is the problem clearly and precisely stated?
2. Simplicity and clarity - Is the algorithm clear? Is there a simpler and clearer algorithm?
3. Algorithm correctness - Is the algorithm correct?
4. Amount of work done - What is the running time of the algorithm in terms of the input size (independent of hardware and programming language)?
5. (Sometimes) amount of space used - How much extra memory space does the algorithm use (here we mean the amount of extra space beyond the size of the input). We will say that an algorithm is *in place* if the amount of extra space is constant with respect to input size.
6. (Sometimes) optimality - can we prove that the algorithm does the best of any algorithm?

We consider the problem of evaluating a polynomial. A precise specification of the problem would be:



**Input:** A polynomial $p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$ and a value $z$. We assume the coefficient are stored in an array $a[0..n]$.

**Output:** The evaluation of $p(x)$ at $z$, i.e. $p(z)$.



**Example:** If the input is $p(x) = x^2 + 2x + 1$ and $z = 3$ then the output should be $p(3) = 16$

## Naive solution

We assume that the coefficients $a_0, a_1, a_2, \ldots, a_n$ are stored in an array $a[0..n]$ of size $n+1$



```
NaiveEvaluation(a, n, z)
  res ← 0
  for i ← 0 to n
    zpoweri ← 1
    for j ← 1 to i
      zpoweri ← zpoweri * z
    res ← res + a[i] * zpoweri
  return res
```

We assume that the coefficients $a_0, a_1, a_2, \ldots, a_n$ are stored in an array $a[0..n]$ of size $n + 1$

```
NaiveEvaluation(a, n, z)
  res ← 0
  for i ← 0 to n
    zpoweri ← 1
    for j ← 1 to i
      zpoweri ← zpoweri * z
    res ← res + a[i] * zpoweri
  return res
```

Is it correct?

We assume that the coefficients $a_0, a_1, a_2, \ldots, a_n$ are stored in an array $a[0..n]$ of size $n + 1$



```
NaiveEvaluation(a, n, z)
  res ← 0
  for i ← 0 to n
    zpoweri ← 1
    for j ← 1 to i
      zpoweri ← zpoweri * z
    res ← res + a[i] * zpoweri
  return res
```





Is it correct? Note: To formally prove correctness you need to use mathematical induction

What do we mean by time?



Suppose that we mean the number of lines of pseudocode executed. The question is still imprecise, as the answer will depend on the size of the input.



Let us denote the size of the input by $n$.

The problem is then to determine the number of lines $T(n)$ executed by our algorithm on a polynomial of degree $n$.



The number of lines executed is

$$\begin{aligned} 2 + \sum_{i=0}^{n}(3 + 2i) &= 2\sum_{i=0}^{n} i + 3(n+1) + 2 \\ &= n^2 + 4n + 5 \end{aligned}$$

What do we mean by time?



Suppose that we mean the number of lines of pseudocode executed. The question is still imprecise, as the answer will depend on the size of the input.



Let us denote the size of the input by $n$.

The problem is then to determine the number of lines $T(n)$ executed by our algorithm on a polynomial of degree $n$.



The number of lines executed is

$$\begin{aligned} 2 + \sum_{i=0}^{n}(3+2i) &= 2\sum_{i=0}^{n} i + 3(n+1) + 2 \\ &= n^2 + 4n + 5 \end{aligned}$$

OK... So what?

What do we mean by time?



Suppose that we mean the number of lines of pseudocode executed. The question is still imprecise, as the answer will depend on the size of the input.



Let us denote the size of the input by $n$.

The problem is then to determine the number of lines $T(n)$ executed by our algorithm on a polynomial of degree $n$.



The number of lines executed is

$$\begin{aligned} 2 + \sum_{i=0}^{n}(3+2i) &= 2\sum_{i=0}^{n} i + 3(n+1) + 2 \\ &= n^2 + 4n + 5 \end{aligned}$$

OK... So what? What if the number was $3n^2 - 10n + 62$???

```
NaiveEvaluation(a, n, z)
  res ← 0
  for i ← 0 to n
    zpoweri ← 1
    for j ← 1 to i
      zpoweri ← zpoweri * z
    res ← res + a[i] * zpoweri
  return res
```

```
BetterEvaluation(a, n, z)
  res ← 0
  zpoweri ← 1
  for i ← 0 to n
    res ← res + a[i] * zpoweri
    zpoweri ← zpoweri * z
  return res
```

```
BetterEvaluation(a, n, z)
  res ← 0
  zpoweri ← 1
  for i ← 0 to n
    res ← res + a[i] * zpoweri
    zpoweri ← zpoweri * z
  return res
```



The number of lines executed is

$$3 + \sum_{i=0}^{n} 3 = 3n + 6$$

```
BetterEvaluation(a, n, z)
  res ← 0
  zpoweri ← 1
  for i ← 0 to n
    res ← res + a[i] * zpoweri
    zpoweri ← zpoweri * z
  return res
```



The number of lines executed is

$$3 + \sum_{i=0}^{n} 3 = 3n + 6$$

which is much less than $n^2 + 4n + 5$

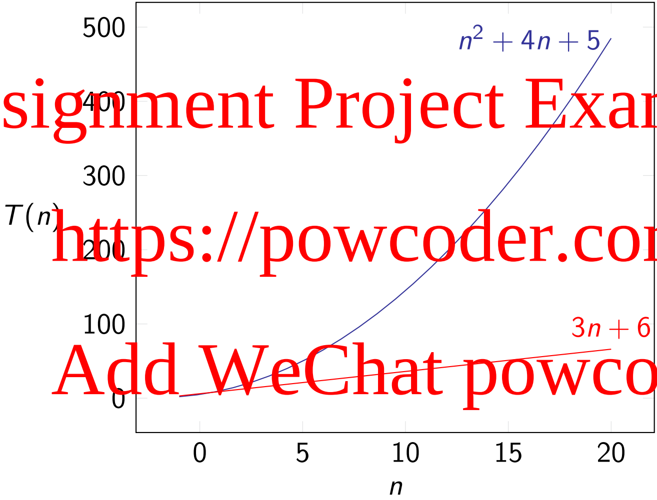
```
BetterEvaluation(a, n, z)
  res ← 0
  zpoweri ← 1
  for i ← 0 to n
    res ← res + a[i] * zpoweri
    zpoweri ← zpoweri * z
  return res
```



The number of lines executed is

$$3 + \sum_{i=0}^{n} 3 = 3n + 6$$

which is much less than $n^2 + 4n + 5$ when $n$ gets large






Note that the comparison of the two functions boils down to the a comparison of the two functions' growth rates

Consider the function $T(n) = n^2 + 4n + 6$ that we produced as the running time of the naive polynomial evaluation algorithm.



We can approximate the behavior of an algorithm by considering only the highest order term in the function T(n). This is because as the size of the problem gets larger, the fastest growing term represents the corresponding growth of the running time.



In this course we will only be interested in this asymptotic behavior of algorithms. This motivates us to define a notation that will make the analysis of algorithms simpler.

We will be interested in three types of asymptotic behavior.



$O$ notation is used to bound the asymptotic behavior of a function from above (upper bound).

**Definition**



Let $f$ and $g$ be (non-negative) functions. If there exist positive constants $c$ and $n_0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$ then we say that $f(n)$ is $O(g(n))$, typically denoted by $f(n) = O(g(n))$.

We will be interested in three types of asymptotic behavior.



$O$ notation is used to bound the asymptotic behavior of a function from above (upper bound).

**Definition**



Let $f$ and $g$ be (non-negative) functions. If there exist positive constants $c$ and $n_0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$ then we say that $f(n)$ is $O(g(n))$, typically denoted by $f(n) = O(g(n))$.



- Does this mean that $f(n)$ is always $\leq g(n)$?

We will be interested in three types of asymptotic behavior.



$O$ notation is used to bound the asymptotic behavior of a function from above (upper bound).

**Definition**



Let $f$ and $g$ be (non-negative) functions. If there exist positive constants $c$ and $n_0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$ then we say that $f(n)$ is $O(g(n))$, typically denoted by $f(n) = O(g(n))$.



- Does this mean that $f(n)$ is always $\leq g(n)$? No!

# Big-O notation

We will be interested in three types of asymptotic behavior.

$O$ notation is used to bound the asymptotic behavior of a function from above (upper bound).



**Definition**

Let $f$ and $g$ be (non-negative) functions. If there exist positive constants $c$ and $n_0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$ then we say that $f(n)$ is $O(g(n))$, typically denoted by $f(n) = O(g(n))$.



- Does this mean that $f(n)$ is always $\leq g(n)$? No!
- Does it mean that $f(n)$ is always $\leq cg(n)$?

# Big-O notation

We will be interested in three types of asymptotic behavior.

$O$ notation is used to bound the asymptotic behavior of a function from above (upper bound).



**Definition**

Let $f$ and $g$ be (non-negative) functions. If there exist positive constants $c$ and $n_0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$ then we say that $f(n)$ is $O(g(n))$, typically denoted by $f(n) = O(g(n))$.



- Does this mean that $f(n)$ is always $\leq g(n)$? No!
- Does it mean that $f(n)$ is always $\leq cg(n)$? No!

We will be interested in three types of asymptotic behavior.



$O$ notation is used to bound the asymptotic behavior of a function from above (upper bound).

**Definition**



Let $f$ and $g$ be (non-negative) functions. If there exist positive constants $c$ and $n_0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$ then we say that $f(n)$ is $O(g(n))$, typically denoted by $f(n) = O(g(n))$.



- Does this mean that $f(n)$ is always $\leq g(n)$? No!
- Does it mean that $f(n)$ is always $\leq cg(n)$? No!
- It means that $f(n)$ is eventually $\leq cg(n)$, for large enough values of $n$.

We will be interested in three types of asymptotic behavior.



$O$ notation is used to bound the asymptotic behavior of a function from above (upper bound).

**Definition**



Let $f$ and $g$ be (non-negative) functions. If there exist positive constants $c$ and $n_0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$ then we say that $f(n)$ is $O(g(n))$, typically denoted by $f(n) = O(g(n))$.



- Does this mean that $f(n)$ is always $\leq g(n)$? No!
- Does it mean that $f(n)$ is always $\leq cg(n)$? No!
- It means that $f(n)$ is eventually $\leq cg(n)$, for large enough values of $n$.
- Big-O notation is the one we will use 95% of the time

Consider the function $T(n) = n^2 + 4n + 5$ that we produced as the running time of the naive polynomial evaluation algorithm.



**Claim**

$T(n) = O(n^2)$



**Proof.**

Choose $c = 10$ and $n_0 = 1$. We verify that for all $n \geq 1$, $n^2 + 4n + 5 \leq 10n^2$:



$$\begin{aligned} n^2 + 4n + 5 &\leq n^2 + 4n^2 + 5n^2 \text{ (since } n \geq 1\text{)} \\ &\leq 10n^2 \end{aligned}$$

$\square$

Consider the function $T(n) = n^2 + 4n + 5$ that we produced as the running time of the naive polynomial evaluation algorithm.



**Claim**

$T(n) = O(n^2)$



**Proof.**

Choose $c = 10$ and $n_0 = 1$. We verify that for all $n \geq 1$, $n^2 + 4n + 5 \leq 10n^2$:



$$\begin{aligned} n^2 + 4n + 5 &\leq n^2 + 4n^2 + 5n^2 \text{ (since } n \geq 1) \\ &\leq 10n^2 \end{aligned}$$

□

Note: We could have chosen $c = 2$ and $n_0 = 5$ but the above choices leads to a simpler argument

Consider the function $T(n) = 3n + 6$ that we produced as the running time of the better polynomial evaluation algorithm.



**Claim.**

$T(n) = O(n)$



**Proof.**

Choose $c = 9$ and $n_0 = 1$. We verify that for all $n \geq 1$, $3n + 6 \leq 9n$:



$$\begin{aligned} 3n + 6 &\leq 3n + 6n \text{ (since } n \geq 1) \\ &\leq 9n \end{aligned}$$

$\square$

Consider the function $T(n) = 3n + 6$ that we produced as the running time of the better polynomial evaluation algorithm.



Claim

$T(n) = O(n)$



Proof.

Choose $c = 9$ and $n_0 = 1$. We verify that for all $n \geq 1$, $3n + 6 \leq 9n$:



$$\begin{aligned} 3n + 6 &\leq 3n + 6n \text{ (since } n \geq 1) \\ &\leq 9n \end{aligned}$$

$\square$

Note: Again, we could have chosen $c = 4$ and $n_0 = 6$ but the above choices leads to a simpler argument

Consider the function $T(n) = 3n + 6$ that we produced as the running time of the better polynomial evaluation algorithm.



**Claim**

$T(n) = O(n^2)$

**Proof.**



Left as an exercise $\square$

Consider again the function $T(n) = n^2 + 4n + 5$ that we produced as the running time of the naive polynomial evaluation algorithm.



**Claim**

$T(n) = O(n)$

Consider again the function $T(n) = n^2 + 4n + 5$ that we produced as the running time of the naive polynomial evaluation algorithm.

**Claim**

$T(n) = O(n)$



No, that is not true!

If it were true, it would mean that there exist positive constants $c$ and $n_0$ such that $n^2 + 4n + 5 \leq cn$ for all $n \geq n_0$.



However, when $n \geq c$ then $n^2 + 4n + 5 \geq cn + 4n + 5 > cn$

So, for $n \geq \max\{n_0, c\}$ we get that $n^2 + 4n + 5 \leq cn$ and that $n^2 + 4n + 5 > cn$ which is a contradiction

$\Omega$ notation is used to bound the asymptotic behavior of a function from below (lower bound).



**Definition**

Let $f$ and $g$ be (non-negative) functions. If there exist positive constants $c$ and $n_0$ such that $f(n) \geq cg(n)$ for all $n \geq n_0$ then we say that $f(n)$ is $\Omega(g(n))$, denoted by $f(n) = \Omega(g(n))$.



It means that $f(n)$ is eventually $\geq cg(n)$.

Consider again the function $T(n) = n^2 + 4n + 5$ that we produced as the running time of the naive polynomial evaluation algorithm.



**Claim**

$T(n) = \Omega(n^2)$.



**Proof.**

Choose $c = 1$ and $n_0 = 1$. We easily verify that for all $n \geq 1$



$$n^2 + 4n + 5 \geq n^2$$

$\square$

So $T(n) = O(n^2)$ and $T(n) = \Omega(n^2)$

$\Theta$ notation is used to capture exactly the asymptotic behavior of a function



**Definition**

Let $f$ and $g$ be (non-negative) functions. If there exist positive constants $c_1$, $c_2$ and $n_0$ such that for all $n \geq n_0$, $c_1 g(n) \leq f(n) \leq c_2 g(n)$ then we say that that $f(n)$ is $\Theta(g(n))$, denoted $f(n) = \Theta(g(n))$.



In other words,



$f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n)$ and $f(n) = \Omega(g(n))$,

and $g(n)$ is said to be an asymptotically tight bound on the function f(n)

**Claim (Transitivity)**

*If $f(n) = O(g(n))$ and $g(n) = O(h(n))$ then $f(n) = O(h(n))$.*

Note: This also holds for $\Omega$ and $\Theta$.



**Proof.**

If $f(n) = O(g(n))$ then there exist $c_1, n_1$ s.t. for all $n \geq n_1$, $f(n) \leq c_1 g(n)$.



If $g(n) = O(h(n))$ then there exist $c_2, n_2$ s.t. for all $n \geq n_2$, $g(n) \leq c_2 h(n)$.

Let $n3 = max\{n1, n2\}$ and let $c_3 = c1 * c2$. Then for all $n \geq n_3$, $f(n) \leq c_1 g(n) \leq c_1 c_2 h(n) = c3h(n)$. Thus $f(n) = O(h(n))$. $\square$

**Claim**

*If* $f_1(n) = O(g_1(n))$, $f_2(n) = O(g_2(n))$, *and* $g_1(n) = O(g_2(n))$ *then* $f_1(n) + f_2(n) = O(g_2(n)))$.



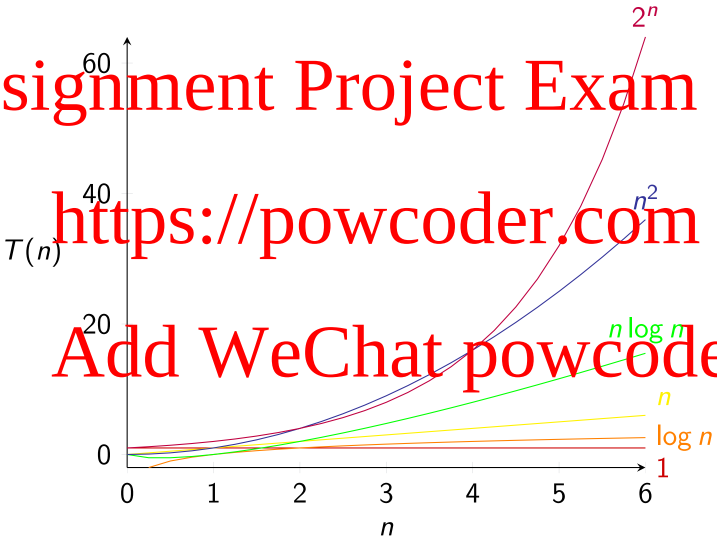
The proof is left as a homework problem.

**Exercise**

*Compare the asymptotic growth rates of the following functions:*



$$1, \lg n, n, n \lg n, n^2, 2^n.$$


$2^n$
$n^2$
$n \log n$
$n$
$\log n$
1
60
40
20
0
$T(n)$
0
1
2
3
4
5
6
$n$

Consider again our algorithm for polynomial evaluation:

```
BetterEvaluation(a, n, z)
  res ← 0
  zpoweri ← 1
  for i ← 0 to n
    res ← res + a[i] * zpoweri
    zpoweri ← zpoweri * z
  return res
```

Consider again our algorithm for polynomial evaluation.



```
BetterEvaluation(a, n, z)
  res ← 0
  zpoweri ← 1
  for i ← 0 to n
    res ← res + a[i] * zpoweri
    zpoweri ← zpoweri * z
  return res
```



Is it optimal?

By rewriting $p(x)$ as

$$\begin{aligned} p(x) &= a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n \\ &= a_0 + x(a_1 + x(a_2 + x(a_3 + x(\ldots(a_{n-1} + x(a_n + 0))\ldots)))) \end{aligned}$$



we develop an alternate algorithm:

```
HornersMethod(a, n, z)
res ← 0
for i ← n downto 0
  res ← res * z + a[i]
output res
```





The number of lines executed by Horner's method is

$$2 + \sum_{i=0}^{n} 2 = 2 + 2(n+1) = 2n + 4$$

which is less than $3n + 6$.

Consider again our algorithm for polynomial evaluation:

```
BetterEvaluation(a, n, z)
  res ← 0
  zpoweri ← 1
  for i ← 0 to n
    res ← res + a[i] * zpoweri
    zpoweri ← zpoweri * z
  return res
```



Is it optimal?

Consider again our algorithm for polynomial evaluation:

```
BetterEvaluation(a, n, z)
  res ← 0
  zpoweri ← 1
  for i ← 0 to n
    res ← res + a[i] * zpoweri
    zpoweri ← zpoweri * z
  return res
```



Is it optimal?

Horner's algorithm is clearly better but the running time of both algorithms is $O(n)$

Because any polynomial evaluation algorithm must run in time $\Omega(n)$ (why?), Horner's and BetterEvaluation algorithms are both asymptotically optimal

A complete description of any algorithm has four components:



- **What:** A precise statement of the problem.
- **How:** A precise description of the algorithm.
- **Why:** A proof that the algorithm is correct, i.e. it solves the problem it is supposed to solve.
- **How fast:** An analysis of the running time of the algorithm, also known as *asymptotic analysis*.

A complete description of any algorithm has four components:

- **What:** A precise statement of the problem.
- **How:** A precise description of the algorithm.
- **Why:** A proof that the algorithm is correct, i.e. it solves the problem it is supposed to solve.
- **How fast:** An analysis of the running time of the algorithm, also known as *asymptotic analysis*.





Problem: Input size is not the only parameter that determines running time

A fundamental problem is sorting an array of numbers.

**Input:** A array $a[0..n-1]$ of $n$ numbers

**Output:** A re-ordering of the numbers in the array such that $a[0] \leq a[1] \leq \cdots \leq a[n-1]$



There are many different ways of approaching the problem

InsertionSort works by repeatedly moving elements into their proper relative positions

The steps of the algorithm are:

1. Move $a[1]$ to the left until all elements to its left are no greater than it



2. Move $a[2]$ to the left until all elements to its left are no greater than it
3. Repeat up until element $a[n-1]$

The steps of the algorithm are:

1. Move $a[1]$ to the left until all elements to its left are no greater than it
2. Move $a[2]$ to the left until all elements to its left are no greater than it
3. Repeat up until element $a[n-1]$





Note: After $i$ iterations, the first $i+1$ elements of the list will be in sorted order

For example, let the input list of numbers be 5, 2, 4, 6, 1, 3. Then the iterations of the algorithm will be the following:



| | | | | | | |
|---|---|---|---|---|---|---|
| Initial order: | 5 | 2 | 4 | 6 | 1 | 3 |
| After 1st iteration: | 2 | 5 | 4 | 6 | 1 | 3 |
| After 2nd iteration: | 2 | 4 | 5 | 6 | 1 | 3 |
| After 3rd iteration: | 2 | 4 | 5 | 6 | 1 | 3 |
| After 4th iteration: | 1 | 2 | 4 | 5 | 6 | 3 |
| After 5th iteration: | 1 | 2 | 3 | 4 | 5 | 6 |

# InsertionSort



If a[0..n-1] is an array of numbers to be sorted, the pseudocode for insertion sort is:



```
InsertionSort(a, n)
for j ← 1 to n-1
  key ← a[j]
  i ← j-1
  while i ≥ 0 and a[i] > key
    a[i+1] ← a[i]
    i ← i-1
  a[i+1] ← key
```

Because we are only interested in the asymptotic behavior of the algorithm, we need only focus on an operation in the innermost loop, e.g. the while loop condition:



$$T(n) = \sum_{i=1}^{n-1} \sum_{?} 1.$$



The first summation counts the number of iterations of the for-loop, and the second summation counts the number of times the while-loop condition is evaluated within a single iteration of the outer for-loop



We know that the for-loop will be executed $n-1$ times.

What about the number of times the while-loop condition will be evaluated?

Suppose that the list contains 1 2 3 4 5 6. Then the execution will look like the following:



Initial order: 1 2 3 4 5 6

Suppose that the list contains 1 2 3 4 5 6. Then the execution will look like the following:



| | | | | | | |
|---|---|---|---|---|---|---|
| Initial order: | 1 | 2 | 3 | 4 | 5 | 6 |
| After 1st iteration: | 1 | 2 | 3 | 4 | 5 | 6 |

Suppose that the list contains 1 2 3 4 5 6. Then the execution will look like the following:



| | | | | | | |
|---|---|---|---|---|---|---|
| Initial order: | 1 | 2 | 3 | 4 | 5 | 6 |
| After 1st iteration: | 1 | 2 | 3 | 4 | 5 | 6 |
| After 2nd iteration: | 1 | 2 | 3 | 4 | 5 | 6 |

Suppose that the list contains 1 2 3 4 5 6. Then the execution will look like the following:



| | | | | | | |
|---|---|---|---|---|---|---|
| Initial order: | 1 | 2 | 3 | 4 | 5 | 6 |
| After 1st iteration: | 1 | 2 | 3 | 4 | 5 | 6 |
| After 2nd iteration: | 1 | 2 | 3 | 4 | 5 | 6 |
| After 3rd iteration: | 1 | 2 | 3 | 4 | 5 | 6 |

Suppose that the list contains 1 2 3 4 5 6. Then the execution will look like the following:



| | | | | | | |
|---|---|---|---|---|---|---|
| Initial order: | 1 | 2 | 3 | 4 | 5 | 6 |
| After 1st iteration: | 1 | 2 | 3 | 4 | 5 | 6 |
| After 2nd iteration: | 1 | 2 | 3 | 4 | 5 | 6 |
| After 3rd iteration: | 1 | 2 | 3 | 4 | 5 | 6 |
| After 4th iteration: | 1 | 2 | 3 | 4 | 5 | 6 |

Suppose that the list contains 1 2 3 4 5 6. Then the execution will look like the following:



| | | | | | | |
|---|---|---|---|---|---|---|
| Initial order: | 1 | 2 | 3 | 4 | 5 | 6 |
| After 1st iteration: | 1 | 2 | 3 | 4 | 5 | 6 |
| After 2nd iteration: | 1 | 2 | 3 | 4 | 5 | 6 |
| After 3rd iteration: | 1 | 2 | 3 | 4 | 5 | 6 |
| After 4th iteration: | 1 | 2 | 3 | 4 | 5 | 6 |
| After 5th iteration: | 1 | 2 | 3 | 4 | 5 | 6 |

Suppose that the list contains 1 2 3 4 5 6. Then the execution will look like the following:



| | | | | | | |
|---|---|---|---|---|---|---|
| Initial order: | 1 | 2 | 3 | 4 | 5 | 6 |
| After 1st iteration: | 1 | 2 | 3 | 4 | 5 | 6 |
| After 2nd iteration: | 1 | 2 | 3 | 4 | 5 | 6 |
| After 3rd iteration: | 1 | 2 | 3 | 4 | 5 | 6 |
| After 4th iteration: | 1 | 2 | 3 | 4 | 5 | 6 |
| After 5th iteration: | 1 | 2 | 3 | 4 | 5 | 6 |





Note that the while-loop condition is evaluated only once in every iteration of the outer for-loop and so

$$T(n) = \sum_{j=1}^{n-1} 1 = O(n)$$

Suppose that the list contains 6 5 4 3 2 1. Then the execution will look like the following:



Initial order: 6 5 4 3 2 1

Suppose that the list contains 6 5 4 3 2 1. Then the execution will look like the following:

| | | | | | | |
|---|---|---|---|---|---|---|
| Initial order: | 6 | 5 | 4 | 3 | 2 | 1 |
| After 1st iteration: | 5 | 6 | 4 | 3 | 2 | 1 |

Suppose that the list contains 6 5 4 3 2 1. Then the execution will look like the following:



| | | | | | | |
|---|---|---|---|---|---|---|
| Initial order: | 6 | 5 | 4 | 3 | 2 | 1 |
| After 1st iteration: | 5 | 6 | 4 | 3 | 2 | 1 |
| After 2nd iteration: | 4 | 5 | 6 | 3 | 2 | 1 |

Suppose that the list contains 6 5 4 3 2 1. Then the execution will look like the following:

| | | | | | | |
|---|---|---|---|---|---|---|
| Initial order: | 6 | 5 | 4 | 3 | 2 | 1 |
| After 1st iteration: | 5 | 6 | 4 | 3 | 2 | 1 |
| After 2nd iteration: | 4 | 5 | 6 | 3 | 2 | 1 |
| After 3rd iteration: | 3 | 4 | 5 | 6 | 2 | 1 |

Suppose that the list contains 6 5 4 3 2 1. Then the execution will look like the following:



| | | | | | | |
|---|---|---|---|---|---|---|
| Initial order: | 6 | 5 | 4 | 3 | 2 | 1 |
| After 1st iteration: | 5 | 6 | 4 | 3 | 2 | 1 |
| After 2nd iteration: | 4 | 5 | 6 | 3 | 2 | 1 |
| After 3rd iteration: | 3 | 4 | 5 | 6 | 2 | 1 |
| After 4th iteration: | 2 | 3 | 4 | 5 | 6 | 1 |

Suppose that the list contains 6 5 4 3 2 1. Then the execution will look like the following:



| | | | | | | |
|---|---|---|---|---|---|---|
| Initial order: | 6 | 5 | 4 | 3 | 2 | 1 |
| After 1st iteration: | 5 | 6 | 4 | 3 | 2 | 1 |
| After 2nd iteration: | 4 | 5 | 6 | 3 | 2 | 1 |
| After 3rd iteration: | 3 | 4 | 5 | 6 | 2 | 1 |
| After 4th iteration: | 2 | 3 | 4 | 5 | 6 | 1 |
| After 5th iteration: | 1 | 2 | 3 | 4 | 5 | 6 |



Within iteration $j$ of the outer for-loop, the while-loop condition is evaluated for $i = j-1, j-2, \ldots, 0$, i.e., a total of $j$ times. Thus



$$T(n) = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2} = O(n^2)$$

The running time of an algorithm can depend on the type of input it is given. Some inputs are "harder" than others for an algorithm. There are three different approaches to analyzing running times:

The running time of an algorithm can depend on the type of input it is given. Some inputs are "harder" than others for an algorithm. There are three different approaches to analyzing running times:



**Best-case** This is the minimum number of steps the algorithm can take on an input of size $n$. It is produced by the inputs on which the algorithm behaves the best. Example 1 is the best case for insertion sort.

The running time of an algorithm can depend on the type of input it is given. Some inputs are "harder" than others for an algorithm. There are three different approaches to analyzing running times:



**Best-case** This is the minimum number of steps the algorithm can take on an input of size $n$. It is produced by the inputs on which the algorithm behaves the best. Example 1 is the best case for insertion sort.



**Worst-case** This represents the maximum number of steps the algorithm can take on an input of size $n$. It provides a guarantee of performance. Example 2 is the worst case for insertion sort.

The running time of an algorithm can depend on the type of input it is given. Some inputs are "harder" than others for an algorithm. There are three different approaches to analyzing running times:



**Best-case** This is the minimum number of steps the algorithm can take on an input of size $n$. It is produced by the inputs on which the algorithm behaves the best. Example 1 is the best case for insertion sort.



**Worst-case** This represents the maximum number of steps the algorithm can take on an input of size $n$. It provides a guarantee of performance. Example 2 is the worst case for insertion sort.



**Average-case** This gives the average (or expected) number of steps over all possible inputs to the algorithm. In order to be computed, a probability distribution on the inputs must be assumed.

The running time of an algorithm can depend on the type of input it is given. Some inputs are "harder" than others for an algorithm. There are three different approaches to analyzing running times:



**Worst-case** This represents the maximum number of steps the algorithm can take on an input of size $n$. It provides a guarantee of performance. Example 2 is the worst case for insertion sort.

If we assume that any ordering is equally likely, then we can argue that we expect that each `a[j]` moves $\frac{1}{2}(j-1)$ positions to the left. In other words, the number of times the while-loop condition is evaluated in each iteration of the outer for-loop is $\frac{1}{2}j$. The expected number of steps is then



$$T(n) = \sum_{j=1}^{n-1} \frac{1}{2} j = \frac{n(n-1)}{4} = O(n^2)$$

Algorithm design techniques that we will use in this class include:

1. divide-and-conquer
2. backtracking
3. dynamic programming
4. greedy





All of the above techniques rely on reductions.

As your textbook says:

*"Reductions are the single most common technique used in designing algorithms.*

*Reducing one problem X to another problem Y means to write an algorithm for X that uses an algorithm for Y as a black box or subroutine.*

*Crucially, the correctness of the resulting algorithm for X cannot depend in any way on how the algorithm for Y works. The only thing we can assume is that the black box solves Y correctly. The inner workings of the black box are simply none of our business; they're somebody else's problem. It's often best to literally think of the black box as functioning purely by magic."*

1. The problem of evaluating a polynomial is reduced to the basic operations of addition and subtraction



2. In InsertionSort, the problem of sorting is reduced to the (easier) problem of inserting a number into an already sorted list



3. In your textbook, the Huntington-Hill algorithm reduces the problem of apportioning Congress to the problem of maintaining a priority queue that supports the operations Insert and ExtractMax.

Recursion is a special type of reduction that reduces a problem instance to one or more simpler instances of the same problem.

All four algorithm design techniques



1. divide-and-conquer
2. backtracking
3. dynamic programming
4. greedy



that we will use in this class can be described using recursion.

A divide-and-conquer algorithm works as follows:

1. If the problem is small enough, solve it directly (and quickly).
2. Otherwise, divide it into subproblems **(Divide)** that your solve recursively ("as a black box functioning purely by magic"!)
3. Combine the solutions to the subproblems into a solution of the original problem **(Conquer)**.

Consider the problem of computing $a^n$:

**Input:** number $a$ and positive integer $n$

**Output:** $a^n$

**Example:** given $a = 1.5$ and $n = 2$, $a^n = 1.5^2 = 2.25$





The obvious algorithm

```
SlowPower(a, n)
  res ← a
  for i ← 2 to n
    res ← res * a
  return res
```

Consider the problem of computing $a^n$:

Input: number $a$ and positive integer $n$

Output: $a^n$

Example: given $a = 1.5$ and $n = 2$, $a^n = 1.5^2 = 2.25$



The obvious algorithm:



```
SlowPower(a, n)
  res ← a
  for i ← 2 to n
    res ← res * a
  return res
```



Running time: $T(n) = O(n)$

Consider the problem of computing $a^n$:

Input: number $a$ and positive integer $n$

Output: $a^n$

Example: given $a = 1.5$ and $n = 2$, $a^n = 1.5^2 = 2.25$



The obvious algorithm

```
SlowPower(a, n)
  res ← a
  for i ← 2 to n
    res ← res * a
  return res
```

Running time: $T(n) = O(n)$

How could divide-and-conquer possibly help?

Indian scholar Pingala proposed the following recursive formula in 2nd century BCE!



$$a^n = \begin{cases} 1 & \text{if } n = 0 \\ (a^{n/2})^2 & \text{if } n > 0 \text{ and n is even} \\ (a^{\lfloor n/2 \rfloor})^2 a & \text{otherwise} \end{cases}$$

which leads to the following exponentiation algorithm:



```
PingalaPower(a, n)
  if n ← 1
    return a
  tmp ← PingalaPower(a, ⌊n/2⌋)
  if n is even
    return tmp*tmp
  else
    return tmp*tmp*a
```

Why is it said to be fast?



```
PingalaPower(a, n)
  if n ← 1
    return a
  tmp ← PingalaPower(a, ⌊n/2⌋)
  if n is even
    return tmp*tmp
  else
    return tmp*tmp*a
```

# Fast exponentiation

Why is it said to be fast?



How many multiplications are performed by this algorithm?

```
PingalaPower(a, n)
  if n ≤ 1
    return a
  tmp ← PingalaPower(a, ⌊n/2⌋)
  if n is even
    return tmp*tmp
  else
    return tmp*tmp*a
```