How $O$ distinguishes between major and incremental improvements

At this point, we have seen

- the Definition of $O$ and related order notations, and have seen
- some simple ways of using the properties of $O$ to analyze the time of algorithms up to order.

Next, we'll look at how we can use order notation to help us design better algorithms. {What is better?}

Order notation naturally distinguishes between small improvements in algorithm design and major gains.
Summing Triple

**Input:** An array \(A[1, \ldots, n]\) of integers.

**Summing Triple:** A **Summing Triple** is a list of three indices \(1 \leq i, j, k \leq n\) so that \(A[i] + A[j] = A[k]\).

**Problem:** Is there a **Summing Triple**?

**Example:** \(A[1,2,3,4] = [3,-6,5,8]\).
Summing Triple: Most obvious algorithm („Brute-Force“)

\[ A[1,2,3,4] = [3,-6,5,8]. \]

**SummingTriple**(\(A[1..n]\))

1. for(\(i:=1..n\))
2. {
3.  for(\(j:=1..n\))
4.  {
5.   for(\(k:=1..n\))
6.   {
8.   } // for(\(k:=1..n\))
9. } // for(\(j:=1..n\))
10 } // for(\(i:=1..n\))
11 return(false);
Summing Triple: Time analysis of the Brute-Force-algorithm

A[1,2,3,4] = [3,-6,5,8].

```plaintext
SummingTriple(A[1..n])
1 for(i:=1..n) 3 Nested Loops, each makes maximal n iterations, so n^3 total operations.
2 { 4 Therefore,
3   for(j:=1..n) 5 total operations.
4     { 6
5       for(k:=1..n) 7
6         { 8
7           if(A[i]+A[j] == A[k]) return(true); 9 Basic Operation
8             } // for(k:=1..n) 10
9             } // for(j:=1..n) 11
10           } // for(i:=1..n)
11 return(false);
```

O(1) Basic Operation
Summing Triple: Eliminating some redundancy

A[1,2,3,4] = [3,-6,5,8].

```plaintext
SummingTriple(A[1..n])
1  for (i:=1..n)
2      {
3          for (j:=i..n)
4              {
5                  for (k:=1..n)
6                      {
8                      } // for(k:=1..n)
9                  } // for(j:=i..n)
10              } // for(i:=1..n)
11  return false;
```

We checked every \( i \) and \( j \) twice, in both orders.

So this new algorithm has eliminated about half of the work of the previous one.

But a constant factor of 1/2 does not change the order, so we still have:

\[ T(n) \in O(n^3) \]

\[ ((n^3+n^2)/2 \text{ exactly}) \]
Summing Triple: Tightness of the bound

\[ A[1,2,3,4] = [3,-6,5,8] \]


```plaintext
SummingTriple(A[1..n])
1   for(i:=1..n)
2     {
3       for(j:=i..n)
4         {
5           for(k:=1..n)
6             {
7               if(A[i]+A[j] == A[k]) return(true);
8             } // for(k:=1..n)
9           } // for(j:=i..n)
10         } // for(i:=1..n)
11     return(false);
```

The innermost \( k \)-loop takes \( n \) steps, and inside we take at least some constant time \( \Omega(1) \), so the inside loop takes \( \Omega(n) \) time each time.
Summing Triple: Tightness of the bound

\[ A[1,2,3,4] = [3,-6,5,8]. \]


```java
SummingTriple(A[1..n])
1  for(i:=1..n)
2   {
3     for(j:=i..n)
4       {
5         for(k:=1..n)
6           {
7             if(A[i]+A[j] == A[k]) return(true);
8             } // for(k:=1..n)
9           } // for(j:=i..n)
10         } // for(i:=1..n)
11 return(false);
```

The innermost \( k \)-loop takes \( n \) steps, and inside we take at least some constant time \( \Omega(1) \), so the inside loop takes \( \Omega(n) \) time each time.

The number of times we go through the \( j \) loop depends on \( i \). But there are \( n/2 \) times (when \( i=1..n/2 \)) when we go through the \( j \)-loop at least \( n/2 \) times. That means the inside \( k \)-loop is executed at least \( n/2 \times n/2 = n^2/4 \) times, and each time it takes \( \Omega(n) \).

Thus the total time is also \( \Omega(n^3) \).
Summing Triple: Tightness of the bound

\[ A[1,2,3,4] = [3,-6,5,8]. \]

The innermost \( k \)-loop takes \( n \) steps, and inside we take at least some constant time \( \Omega(1) \), so the inside loop takes \( \Omega(n) \) time each time. The number of times we go through the \( j \) loop depends on \( i \). But there are \( n/2 \) times (when \( i = 1..n/2 \)) when we go through the \( j \)-loop at least \( n/2 \) times. That means the inside \( k \)-loop is executed at least \( n/2 \times n/2 = n^2/4 \) times, and each time it takes \( \Omega(n) \). Thus the total time is also \( \Omega(n^3) \).

Since this algorithm is both \( O(n^3) \) and \( \Omega(n^3) \), its time is \( \Theta(n^3) \), so our original analysis was tight.
Summing Triple: Viewing the algorithm more conceptually

Here's another way of describing the same algorithm:

For each $1 \leq i \leq j \leq n$, we use linear search to see if $A[i] + A[j]$ is in the array $A[1..n]$.

It doesn't change the algorithm, but this different viewpoint suggests a possibility for improvement.
Summing Triple: Linear Search

A[1,2,3,4] = [-6,3,5,8].

```plaintext
SortedListSummingTriple(A[1..n]:
    sorted array of integers)

1 for (i:=1..n)
2 {
3     for (j:=i..n)
4         {
5             if (LinearSearch(A,A[i]+A[j]) return(true);
6         } // for (j:=i..n)
7     } // for (i:=1..n)
8 return(false);
```

Since **Linear Search** takes $O(n)$ time, and we have two nested loops with at most / fewer than $n$ iterations each, the total time is $O(n^3)$.
Summing Triple: Binary Search

A[1,2,3,4] = [-6,3,5,8].

```
function SortedSummingTriple(A[1..n]:
                           sorted array of integers)
1  for(i:=1..n)
2    {                  // for(i:=1..n)
3      for(j:=i..n)
4        {             // for(j:=i..n)
5          if (BinarySearch(A,A[i]+A[j])) return(true);
6        } // for(j:=i..n)
7    } // for(i:=1..n)
8  return(false);
```

Since **Binary Search** takes $O(\log n)$ time, and we have two nested loops with at most / fewer than $n$ iterations each, the total time is $O(n^2 \log n)$.
We cannot *assume* the array A is sorted, but we can *ensure* that it is sorted:

```python
SummingTriple(A[1..n]: array of integers)
1 BubbleSort(A);
2 return(SortedSummingTriple(A));
```

How much time does `SummingTriple` take? Is it better or worse than $O(n^3)$?
Summing Triple: Time analysis

We cannot *assume* the array $A$ is sorted, but we can *ensure* that it is sorted:

```
SummingTriple(A[1..n]: array of integers)
1 BubbleSort(A);
2 return(SortedSummingTriple(A));
```

How much time does $\texttt{SummingTriple}$ take? Is it better or worse than $O(n^3)$?

The new algorithm has two unnested parts, sorting and then using $\texttt{SortedSummingTriple}$. We've already analyzed the two parts. $\texttt{BubbleSort}$ takes time $O(n^2)$, and $\texttt{SortedSummingTriple}$ takes time $O(n^2 \log n)$. So the total time is $O(n^2 + n^2 \log n) = O(n^2 \log n)$. 
We cannot *assume* the array A is sorted, but we can *ensure* that it is sorted:

```plaintext
SummingTriple(A[1..n]: array of integers)
1 BubbleSort(A);
2 return(SortedSummingTriple(A));
```

Because *Bubble Sort*'s time $O(n^2)$ is $o$ of the total time $O(n^2 \log n)$, a better sorting procedure will not improve the total time significantly.

Because $n^2 \log n \in o(n^3)$, this is an asymptotically strictly better algorithm than what we started with.

The best algorithms known for *Summing Triple* take $O(n^2)$ time. The question of whether there is a better algorithm than that is unknown, and the subject of active research.
Intersection
Intersection problem

The problem:
Given two sorted arrays A[1..n] and B[1..n], determine if they intersect, i.e. if there are i, j, such that A[i] = B[j].

A solution:
• We use a linear search to see if B[1] is anywhere in A.
• In general, since B[j] ≥ B[j-1], we can start the search for the next B[j] where our search for B[j-1] left off.
Intersection problem

**Intersect**(A[1..n], B[1..n])

1. `i:=1;
2. **for**`(*j*:=1..n)
3. `{ 
4. **while**(*B*[j] > *A*[i] and *i* ≤ *n*) `*i*++;
5. **if**(*i* > *n*) `**return**(false);
6. **if**(*B*[j] == *A*[i]) `**return**(true);
7. }
8. `**return**(false);

In the worst-case, the inside **while**-loop can run *n* times, and the outside **for**-loop has *n* iterations.

Thus, we have an upper bound of \(O(n^2)\) time.
Intersection problem

\begin{algorithm}
\textbf{Intersect}(A[1..n], B[1..n])
1 \hspace{1em} i := 1;
2 \hspace{1em} \textbf{for}(j := 1..n)
3 \hspace{1em} \{ \\
4 \hspace{2em} \textbf{while}(B[j] > A[i] \textbf{ and } i \leq n) \hspace{1em} i++; \\
5 \hspace{2em} \textbf{if}(i > n) \hspace{1em} \textbf{return}(false); \\
6 \hspace{2em} \textbf{if}(B[j] == A[i]) \hspace{1em} \textbf{return}(true); \\
7 \hspace{1em} \} \\
8 \hspace{1em} \textbf{return}(false);
\end{algorithm}

In the worst-case, the inside \textbf{while}-loop can run \( n \) times, and the outside \textbf{for}-loop has \( n \) iterations.

Thus, we have an upper bound of \( O(n^2) \) time.

\textbf{But this isn't tight.}
Intersection problem

```
Intersect(A[1..n], B[1..n])
1  i:=1;
2  for(j:=1..n)
3    {
4      while(B[j] > A[i] and i ≤ n) i++;
5      if( i > n ) return(false);
6      if( B[j] == A[i] ) return(true);
7    }
8  return(false);
```

In the worst-case, the inside `while`-loop can run \( n \) times, and the outside `for`-loop has \( n \) iterations.

Thus, we have an upper bound of \( O(n^2) \) time.

But this isn't tight.

The inside `while`-loop can run \( n \) times ONCE, but then the rest of the time, it won't be done at all. In fact, except for the last iteration in every `for`-loop, every time line 4 is executed, \( i \) is incremented, and if \( i \) reaches \( n+1 \), the program terminates.

So line 4 only is run \( 2n \) times total, which makes the entire time for this algorithm \( O(n) \).