CS 375 – Analysis of Algorithms

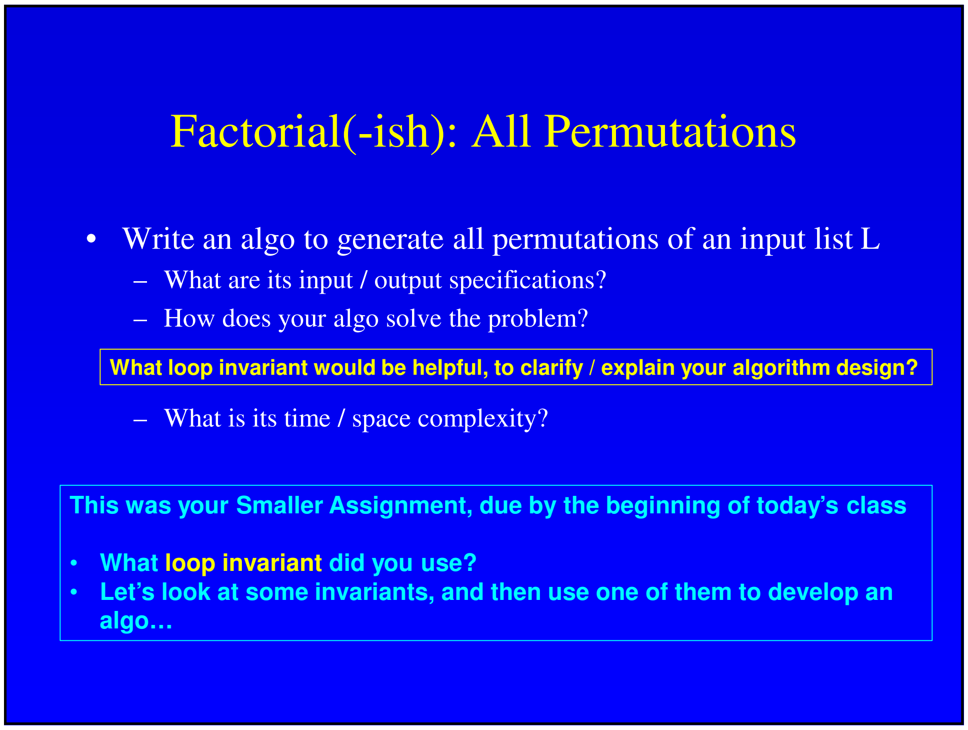
Professor Eric Aaron

Lecture – M W 1:00pm

Lecture Meeting Location: Davis 117

Business

- Thank you to those who participated last Wednesday!
  - I'll look over feedback and have a few words to say on Monday
- Grading update:
  - PS1 update—grades probably returned by end of day Friday
  - Project1 started—grades probably not returned until weekend
    - (Please let me know if you'd like feedback sooner!)
- Coming soon, as CS375 assignments…
  - SA3 likely out today, due Monday
  - PS2 out soon, likely due date Oct. 24
  - Proj2 out soon, likely due date in early November
- Revisions on SA1 (where applicable):
  - This is foundational material
  - Please get to them soon, to help with learning for the course



Factorial(-ish): All Permutations

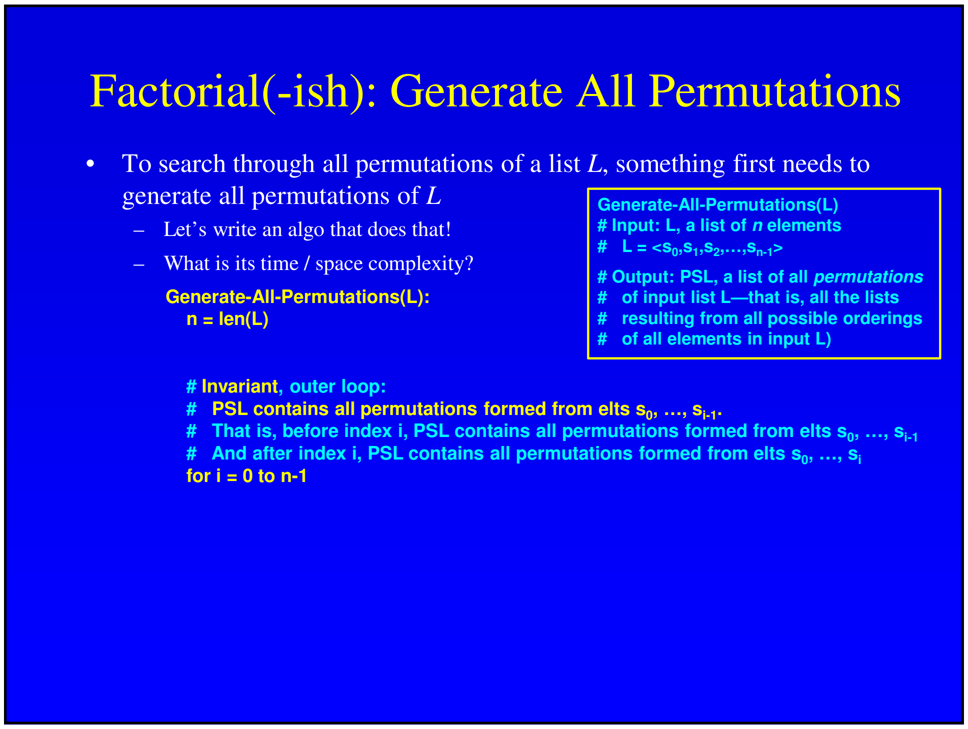
- Write an algo to generate all permutations of an input list L
  - What are its input / output specifications?
  - How does your algo solve the problem?

What loop invariant would be helpful, to clarify / explain your algorithm design?

  - What is its time / space complexity?

This was your Smaller Assignment, due by the beginning of today's class

- **What loop invariant did you use?**
- **Let's look at some invariants, and then use one of them to develop an algo…**

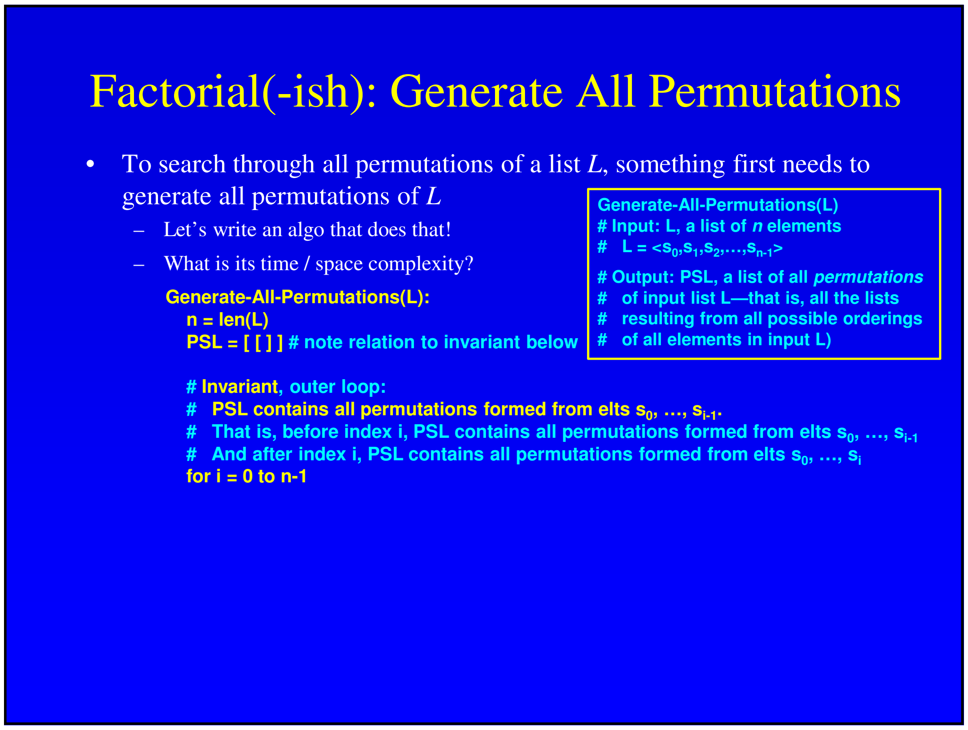


Factorial(-ish): Generate All Permutations

- To search through all permutations of a list *L*, something first needs to generate all permutations of *L*
  - Let's write an algo that does that!
  - What is its time / space complexity?

```
Generate-All-Permutations(L)
# Input: L, a list of n elements
#   L = <$s_0$,$s_1$,$s_2$,…,$s_{n-1}$>
# Output: PSL, a list of all permutations
#   of input list L—that is, all the lists
#   resulting from all possible orderings
#   of all elements in input L)
```

```
Generate-All-Permutations(L):
    n = len(L)

    # Invariant, outer loop:
    #   PSL contains all permutations formed from elts $s_0$, …, $s_{i-1}$.
    #   That is, before index i, PSL contains all permutations formed from elts $s_0$, …, $s_{i-1}$
    #   And after index i, PSL contains all permutations formed from elts $s_0$, …, $s_i$
    for i = 0 to n-1
```

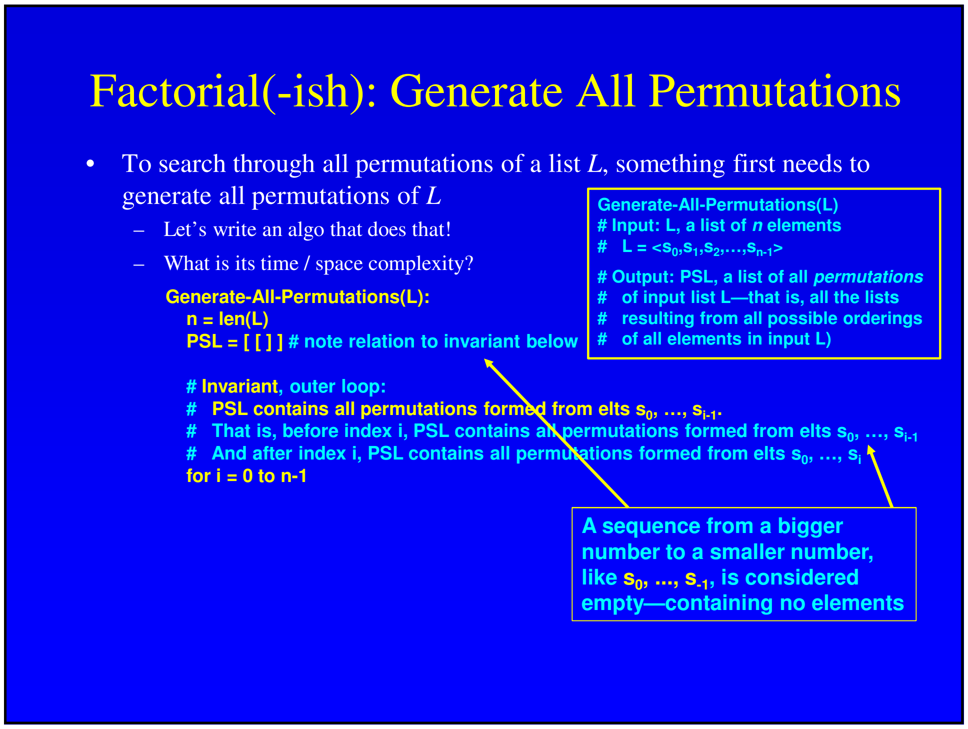


Factorial(-ish): Generate All Permutations

- To search through all permutations of a list *L*, something first needs to generate all permutations of *L*
  - Let's write an algo that does that!
  - What is its time / space complexity?

```
Generate-All-Permutations(L)
# Input: L, a list of n elements
#   L = <$s_0,s_1,s_2,\ldots,s_{n-1}$>
# Output: PSL, a list of all permutations
#   of input list L—that is, all the lists
#   resulting from all possible orderings
#   of all elements in input L)
```

```
Generate-All-Permutations(L):
    n = len(L)
    PSL = [ [ ] ] # note relation to invariant below

    # Invariant, outer loop:
    #  PSL contains all permutations formed from elts $s_0, \ldots, s_{i-1}$.
    #  That is, before index i, PSL contains all permutations formed from elts $s_0, \ldots, s_{i-1}$
    #  And after index i, PSL contains all permutations formed from elts $s_0, \ldots, s_i$
    for i = 0 to n-1
```



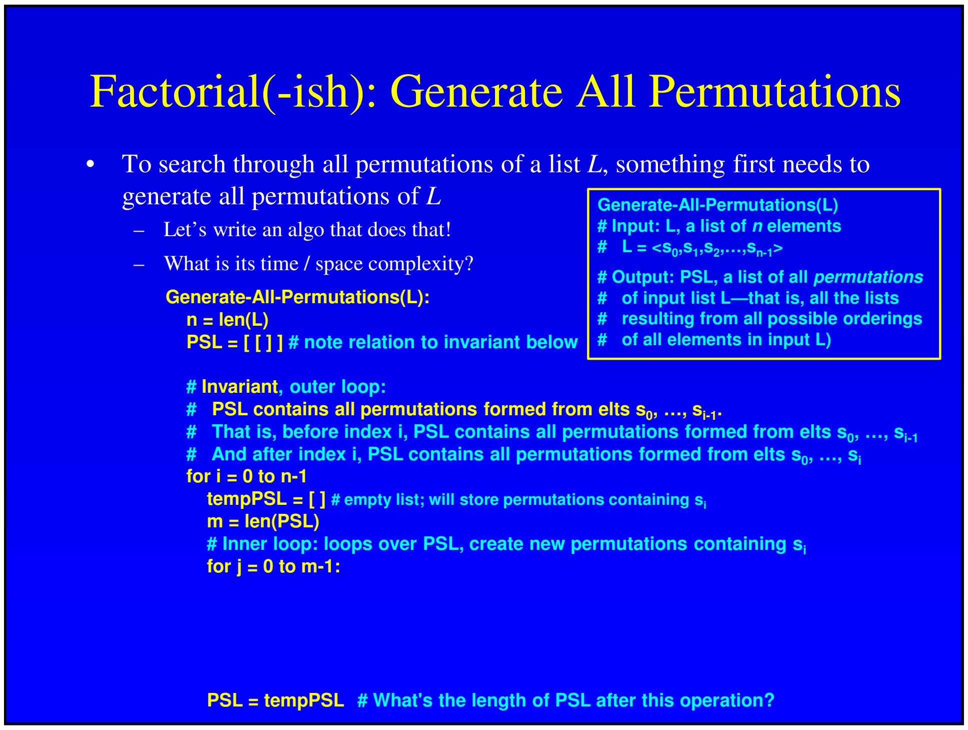
Factorial(-ish): Generate All Permutations

- To search through all permutations of a list *L*, something first needs to generate all permutations of *L*
  - Let's write an algo that does that!
  - What is its time / space complexity?

```
Generate-All-Permutations(L)
# Input: L, a list of n elements
#   L = <$s_0,s_1,s_2,\ldots,s_{n-1}$>
# Output: PSL, a list of all permutations
#   of input list L—that is, all the lists
#   resulting from all possible orderings
#   of all elements in input L)
```

```
Generate-All-Permutations(L):
    n = len(L)
    PSL = [ [ ] ] # note relation to invariant below

    # Invariant, outer loop:
    #  PSL contains all permutations formed from elts $s_0, \ldots, s_{i-1}$.
    #  That is, before index i, PSL contains all permutations formed from elts $s_0, \ldots, s_{i-1}$
    #  And after index i, PSL contains all permutations formed from elts $s_0, \ldots, s_i$
    for i = 0 to n-1
```

A sequence from a bigger number to a smaller number, like $s_0, \ldots, s_{-1}$, is considered empty—containing no elements

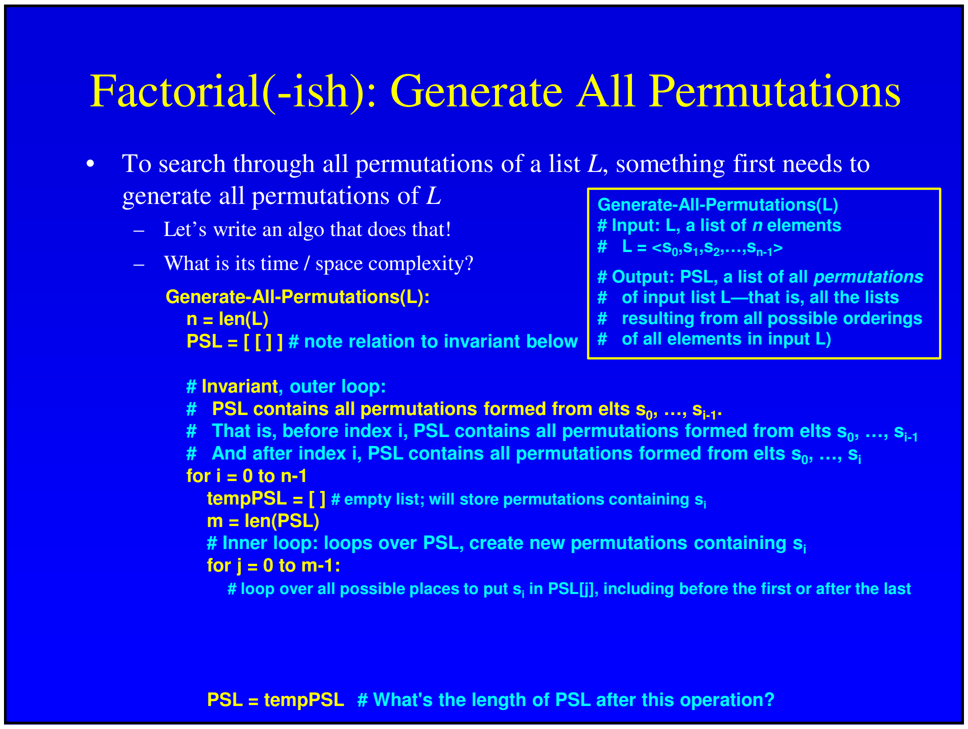


Factorial(-ish): Generate All Permutations

- To search through all permutations of a list *L*, something first needs to generate all permutations of *L*
  - Let's write an algo that does that!
  - What is its time / space complexity?

```
Generate-All-Permutations(L)
# Input: L, a list of n elements
#   L = <$s_0$,$s_1$,$s_2$,…,$s_{n-1}$>
# Output: PSL, a list of all permutations
#   of input list L—that is, all the lists
#   resulting from all possible orderings
#   of all elements in input L)
```

```
Generate-All-Permutations(L):
    n = len(L)
    PSL = [ [ ] ] # note relation to invariant below

    # Invariant, outer loop:
    #   PSL contains all permutations formed from elts $s_0$, …, $s_{i-1}$.
    #   That is, before index i, PSL contains all permutations formed from elts $s_0$, …, $s_{i-1}$
    #   And after index i, PSL contains all permutations formed from elts $s_0$, …, $s_i$
    for i = 0 to n-1
        tempPSL = [ ] # empty list; will store permutations containing $s_i$
        m = len(PSL)
        # Inner loop: loops over PSL, create new permutations containing $s_i$
        for j = 0 to m-1:



        PSL = tempPSL   # What's the length of PSL after this operation?
```

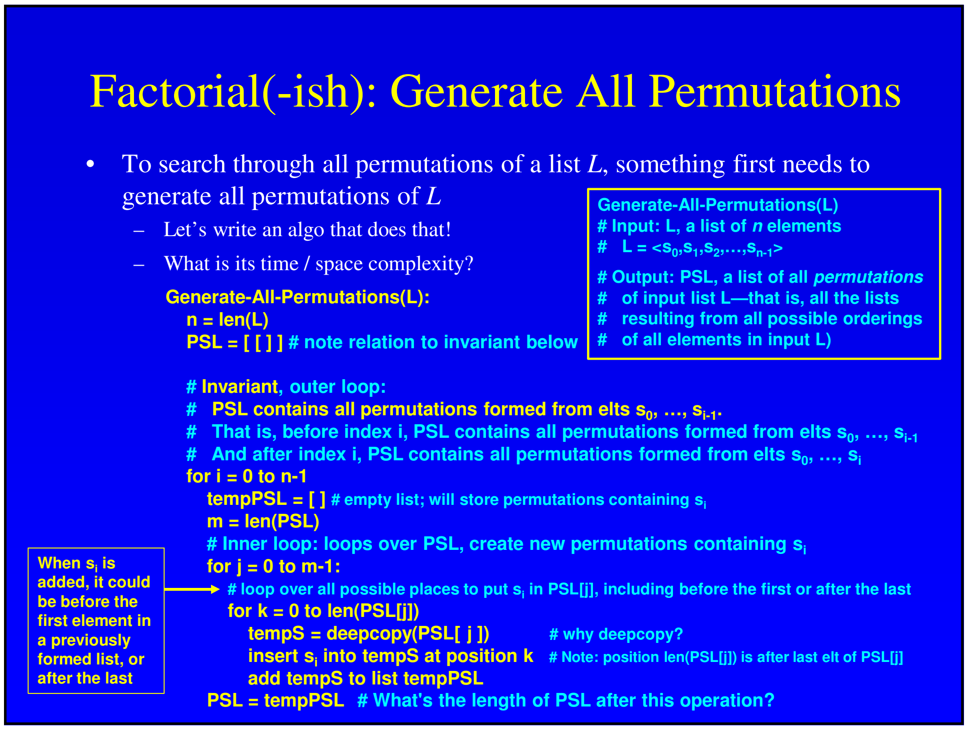


Factorial(-ish): Generate All Permutations

- To search through all permutations of a list *L*, something first needs to generate all permutations of *L*
  - Let's write an algo that does that!
  - What is its time / space complexity?

```
Generate-All-Permutations(L)
# Input: L, a list of n elements
#   L = <$s_0$,$s_1$,$s_2$,…,$s_{n-1}$>
# Output: PSL, a list of all permutations
#   of input list L—that is, all the lists
#   resulting from all possible orderings
#   of all elements in input L)
```

```
Generate-All-Permutations(L):
    n = len(L)
    PSL = [ [ ] ] # note relation to invariant below

    # Invariant, outer loop:
    #   PSL contains all permutations formed from elts $s_0$, …, $s_{i-1}$.
    #   That is, before index i, PSL contains all permutations formed from elts $s_0$, …, $s_{i-1}$
    #   And after index i, PSL contains all permutations formed from elts $s_0$, …, $s_i$
    for i = 0 to n-1
        tempPSL = [ ] # empty list; will store permutations containing $s_i$
        m = len(PSL)
        # Inner loop: loops over PSL, create new permutations containing $s_i$
        for j = 0 to m-1:
            # loop over all possible places to put $s_i$ in PSL[j], including before the first or after the last



        PSL = tempPSL   # What's the length of PSL after this operation?
```



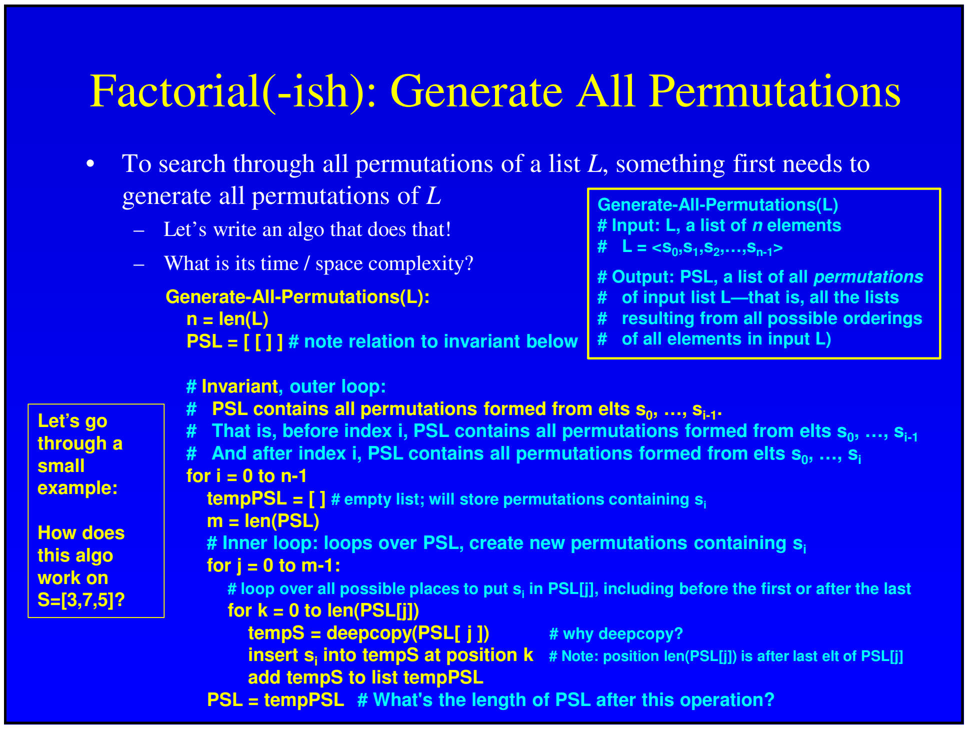
Factorial(-ish): Generate All Permutations

- To search through all permutations of a list *L*, something first needs to generate all permutations of *L*
  - Let's write an algo that does that!
  - What is its time / space complexity?

```
Generate-All-Permutations(L)
# Input: L, a list of n elements
#   L = <s_0,s_1,s_2,…,s_{n-1}>
# Output: PSL, a list of all permutations
#   of input list L—that is, all the lists
#   resulting from all possible orderings
#   of all elements in input L)
```

```
Generate-All-Permutations(L):
  n = len(L)
  PSL = [ [ ] ] # note relation to invariant below

  # Invariant, outer loop:
  #  PSL contains all permutations formed from elts s_0, …, s_{i-1}.
  #  That is, before index i, PSL contains all permutations formed from elts s_0, …, s_{i-1}
  #  And after index i, PSL contains all permutations formed from elts s_0, …, s_i
  for i = 0 to n-1
    tempPSL = [ ] # empty list; will store permutations containing s_i
    m = len(PSL)
    # Inner loop: loops over PSL, create new permutations containing s_i
    for j = 0 to m-1:
      # loop over all possible places to put s_i in PSL[j], including before the first or after the last
      for k = 0 to len(PSL[j])
        tempS = deepcopy(PSL[ j ])          # why deepcopy?
        insert s_i into tempS at position k  # Note: position len(PSL[j]) is after last elt of PSL[j]
        add tempS to list tempPSL
    PSL = tempPSL  # What's the length of PSL after this operation?
```

When s_i is added, it could be before the first element in a previously formed list, or after the last



Factorial(-ish): Generate All Permutations

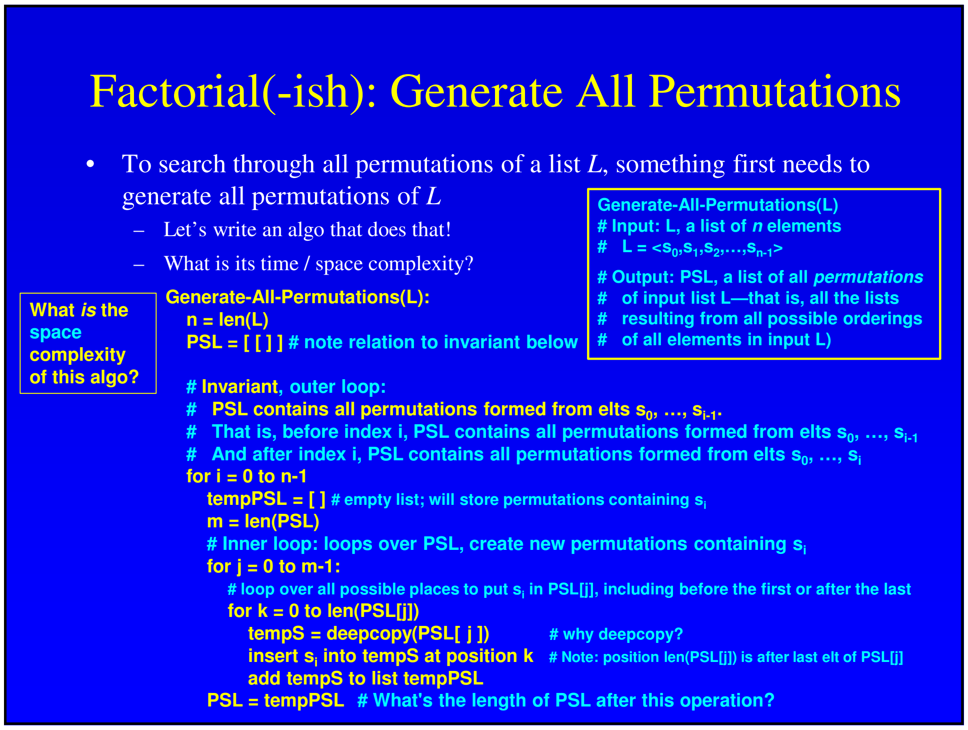
- To search through all permutations of a list *L*, something first needs to generate all permutations of *L*
  - Let's write an algo that does that!
  - What is its time / space complexity?

```
Generate-All-Permutations(L)
# Input: L, a list of n elements
#   L = <s_0,s_1,s_2,…,s_{n-1}>
# Output: PSL, a list of all permutations
#   of input list L—that is, all the lists
#   resulting from all possible orderings
#   of all elements in input L)
```

```
Generate-All-Permutations(L):
  n = len(L)
  PSL = [ [ ] ] # note relation to invariant below

  # Invariant, outer loop:
  #  PSL contains all permutations formed from elts s_0, …, s_{i-1}.
  #  That is, before index i, PSL contains all permutations formed from elts s_0, …, s_{i-1}
  #  And after index i, PSL contains all permutations formed from elts s_0, …, s_i
  for i = 0 to n-1
    tempPSL = [ ] # empty list; will store permutations containing s_i
    m = len(PSL)
    # Inner loop: loops over PSL, create new permutations containing s_i
    for j = 0 to m-1:
      # loop over all possible places to put s_i in PSL[j], including before the first or after the last
      for k = 0 to len(PSL[j])
        tempS = deepcopy(PSL[ j ])          # why deepcopy?
        insert s_i into tempS at position k  # Note: position len(PSL[j]) is after last elt of PSL[j]
        add tempS to list tempPSL
    PSL = tempPSL  # What's the length of PSL after this operation?
```

Let's go through a small example:

How does this algo work on S=[3,7,5]?



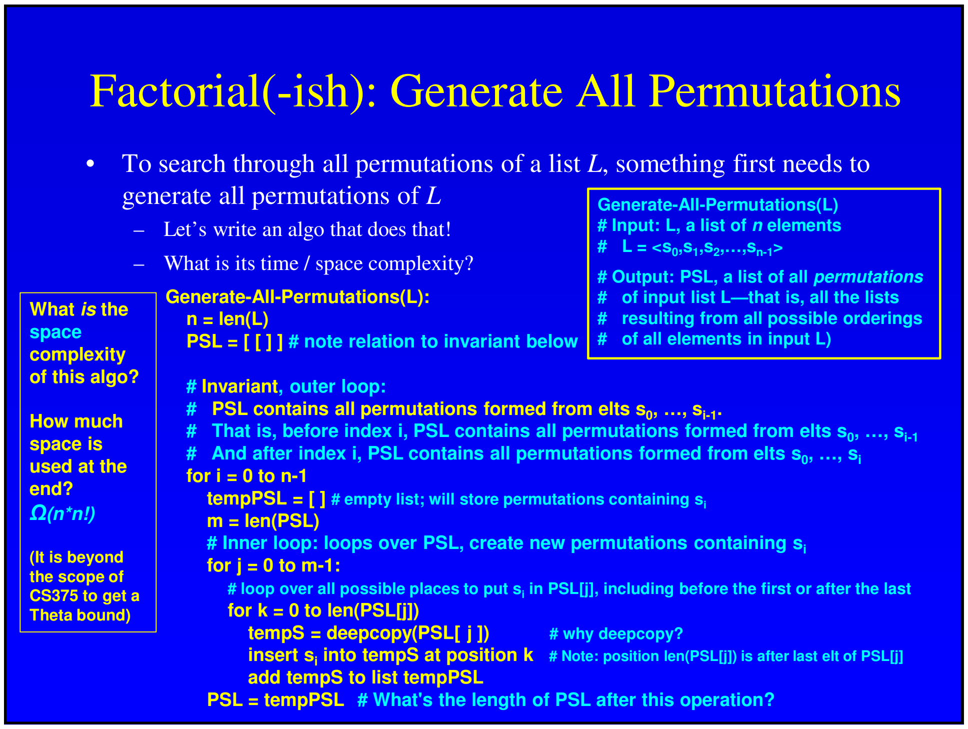
Factorial(-ish): Generate All Permutations

- To search through all permutations of a list *L*, something first needs to generate all permutations of *L*
  - Let's write an algo that does that!
  - What is its time / space complexity?

```
Generate-All-Permutations(L)
# Input: L, a list of n elements
#   L = <$s_0,s_1,s_2,\ldots,s_{n-1}$>
# Output: PSL, a list of all permutations
#   of input list L—that is, all the lists
#   resulting from all possible orderings
#   of all elements in input L)
```

What *is* the space complexity of this algo?

```
Generate-All-Permutations(L):
  n = len(L)
  PSL = [ [ ] ] # note relation to invariant below

  # Invariant, outer loop:
  #  PSL contains all permutations formed from elts $s_0, \ldots, s_{i-1}$.
  #  That is, before index i, PSL contains all permutations formed from elts $s_0, \ldots, s_{i-1}$
  #  And after index i, PSL contains all permutations formed from elts $s_0, \ldots, s_i$
  for i = 0 to n-1
    tempPSL = [ ] # empty list; will store permutations containing $s_i$
    m = len(PSL)
    # Inner loop: loops over PSL, create new permutations containing $s_i$
    for j = 0 to m-1:
      # loop over all possible places to put $s_i$ in PSL[j], including before the first or after the last
      for k = 0 to len(PSL[j])
        tempS = deepcopy(PSL[ j ])          # why deepcopy?
        insert $s_i$ into tempS at position k   # Note: position len(PSL[j]) is after last elt of PSL[j]
        add tempS to list tempPSL
    PSL = tempPSL  # What's the length of PSL after this operation?
```



Factorial(-ish): Generate All Permutations

- To search through all permutations of a list *L*, something first needs to generate all permutations of *L*
  - Let's write an algo that does that!
  - What is its time / space complexity?

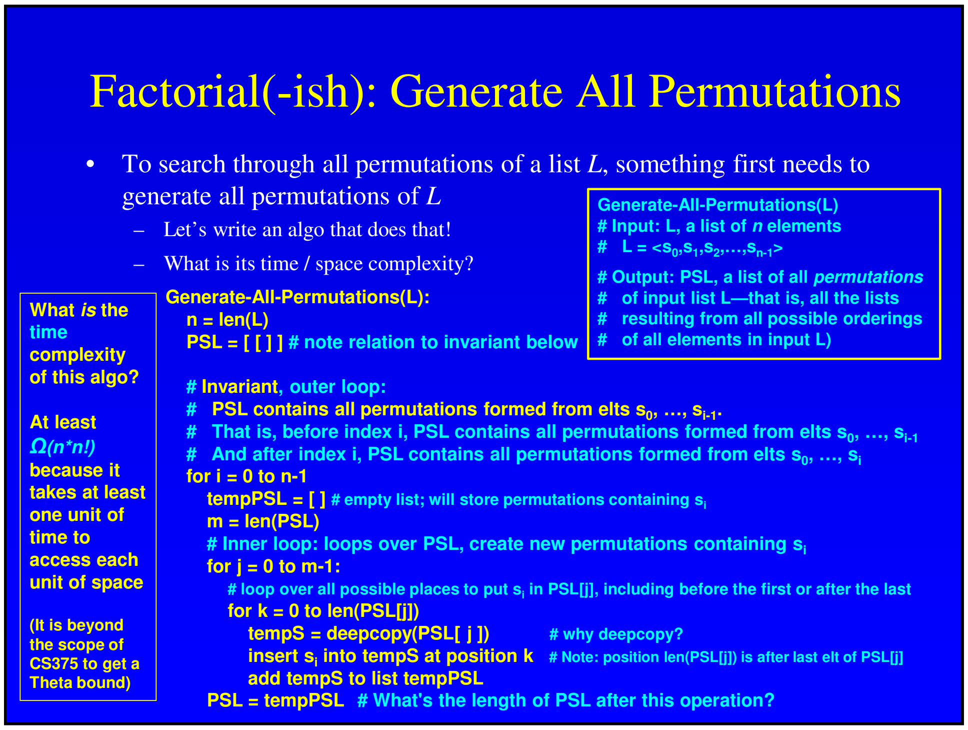
```
Generate-All-Permutations(L)
# Input: L, a list of n elements
#   L = <$s_0,s_1,s_2,\ldots,s_{n-1}$>
# Output: PSL, a list of all permutations
#   of input list L—that is, all the lists
#   resulting from all possible orderings
#   of all elements in input L)
```

What *is* the space complexity of this algo?

How much space is used at the end?
$\Omega(n*n!)$

(It is beyond the scope of CS375 to get a Theta bound)

```
Generate-All-Permutations(L):
  n = len(L)
  PSL = [ [ ] ] # note relation to invariant below

  # Invariant, outer loop:
  #  PSL contains all permutations formed from elts $s_0, \ldots, s_{i-1}$.
  #  That is, before index i, PSL contains all permutations formed from elts $s_0, \ldots, s_{i-1}$
  #  And after index i, PSL contains all permutations formed from elts $s_0, \ldots, s_i$
  for i = 0 to n-1
    tempPSL = [ ] # empty list; will store permutations containing $s_i$
    m = len(PSL)
    # Inner loop: loops over PSL, create new permutations containing $s_i$
    for j = 0 to m-1:
      # loop over all possible places to put $s_i$ in PSL[j], including before the first or after the last
      for k = 0 to len(PSL[j])
        tempS = deepcopy(PSL[ j ])          # why deepcopy?
        insert $s_i$ into tempS at position k   # Note: position len(PSL[j]) is after last elt of PSL[j]
        add tempS to list tempPSL
    PSL = tempPSL  # What's the length of PSL after this operation?
```



Factorial(-ish): Generate All Permutations

- To search through all permutations of a list *L*, something first needs to generate all permutations of *L*
  - Let's write an algo that does that!
  - What is its time / space complexity?

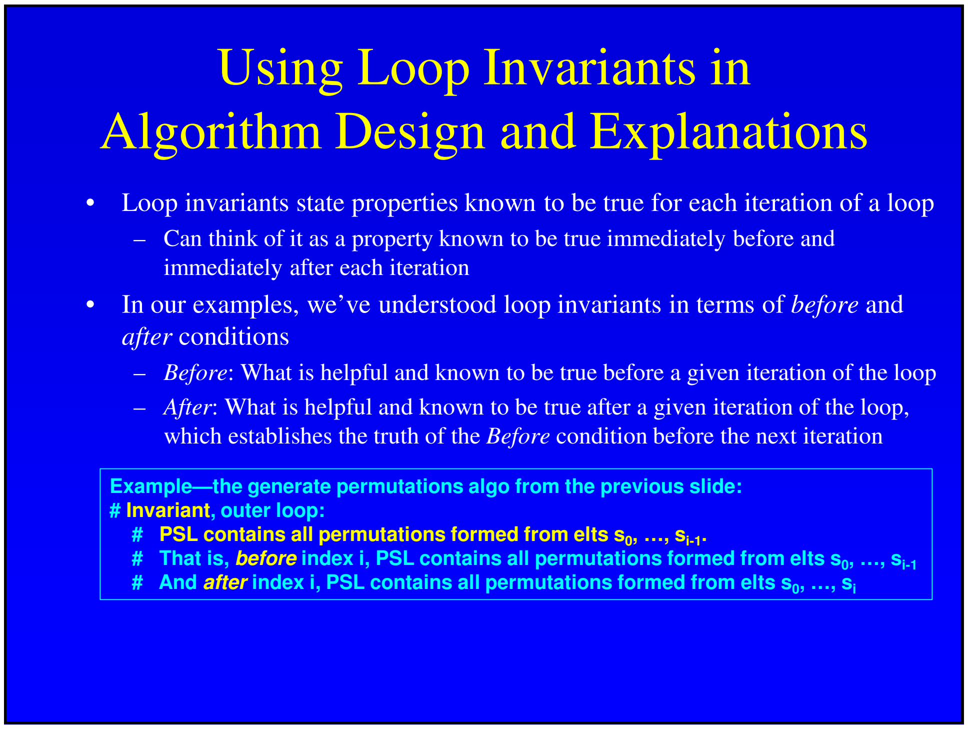
```
Generate-All-Permutations(L)
# Input: L, a list of n elements
#   L = <$s_0$,$s_1$,$s_2$,…,$s_{n-1}$>
# Output: PSL, a list of all permutations
#   of input list L—that is, all the lists
#   resulting from all possible orderings
#   of all elements in input L)
```

What *is* the time complexity of this algo?

At least *Ω(n*n!)* because it takes at least one unit of time to access each unit of space

(It is beyond the scope of CS375 to get a Theta bound)

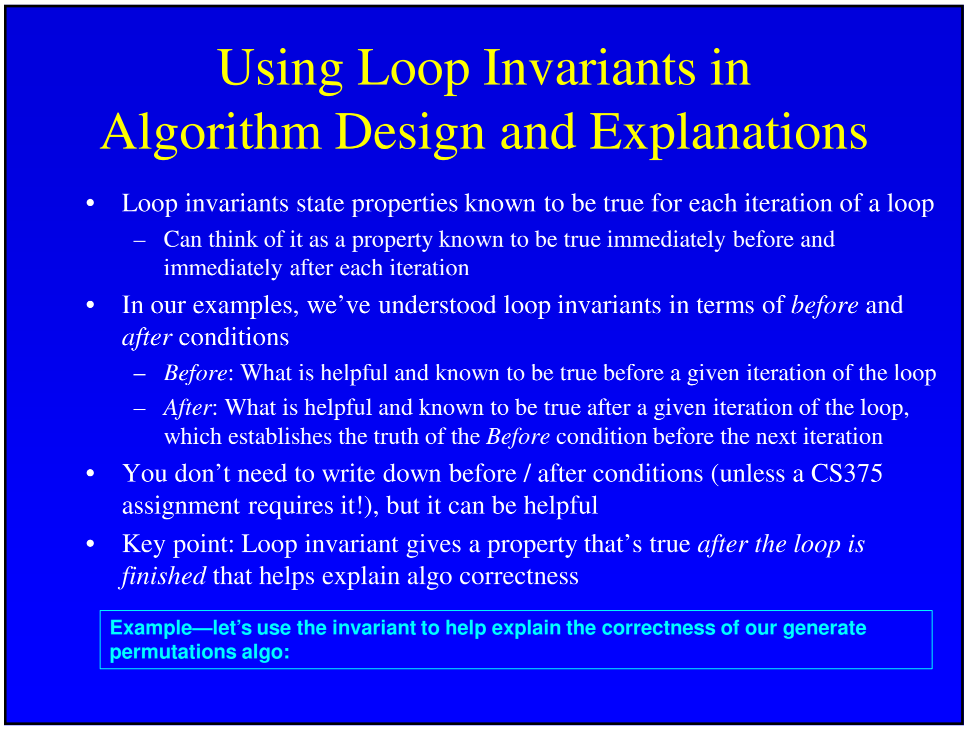
```
Generate-All-Permutations(L):
  n = len(L)
  PSL = [ [ ] ] # note relation to invariant below

  # Invariant, outer loop:
  #   PSL contains all permutations formed from elts $s_0$, …, $s_{i-1}$.
  #   That is, before index i, PSL contains all permutations formed from elts $s_0$, …, $s_{i-1}$
  #   And after index i, PSL contains all permutations formed from elts $s_0$, …, $s_i$
  for i = 0 to n-1
    tempPSL = [ ] # empty list; will store permutations containing $s_i$
    m = len(PSL)
    # Inner loop: loops over PSL, create new permutations containing $s_i$
    for j = 0 to m-1:
      # loop over all possible places to put $s_i$ in PSL[j], including before the first or after the last
      for k = 0 to len(PSL[j])
        tempS = deepcopy(PSL[ j ])          # why deepcopy?
        insert $s_i$ into tempS at position k   # Note: position len(PSL[j]) is after last elt of PSL[j]
        add tempS to list tempPSL
    PSL = tempPSL  # What's the length of PSL after this operation?
```



Using Loop Invariants in Algorithm Design and Explanations

- Loop invariants state properties known to be true for each iteration of a loop
  - Can think of it as a property known to be true immediately before and immediately after each iteration
- In our examples, we've understood loop invariants in terms of *before* and *after* conditions
  - *Before*: What is helpful and known to be true before a given iteration of the loop
  - *After*: What is helpful and known to be true after a given iteration of the loop, which establishes the truth of the *Before* condition before the next iteration

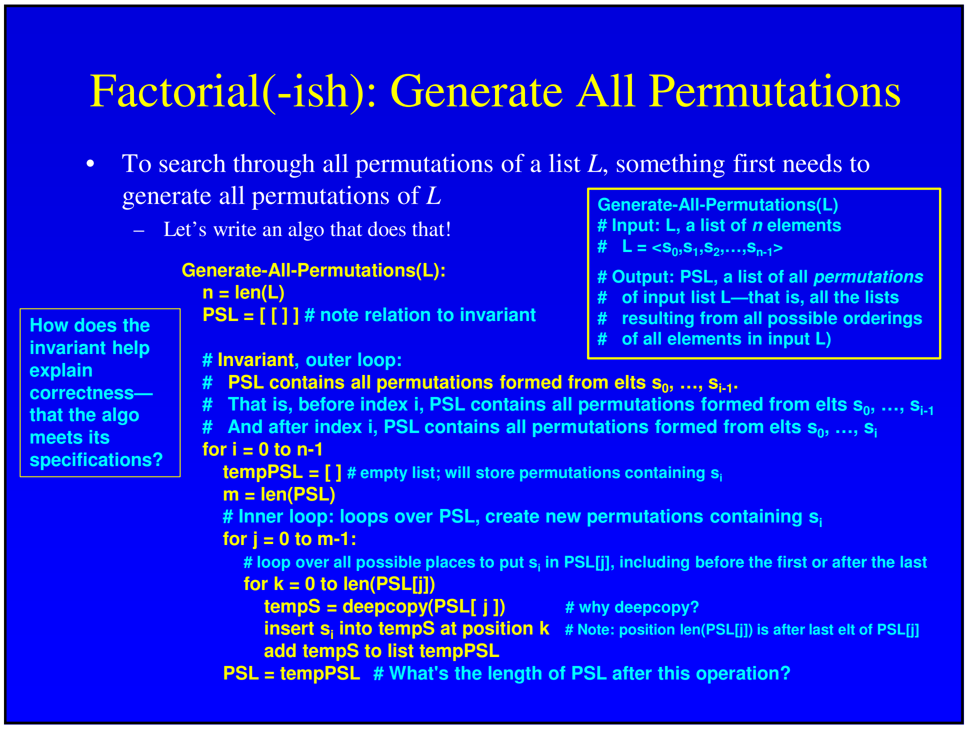
Example—the generate permutations algo from the previous slide:
# Invariant, outer loop:
  # PSL contains all permutations formed from elts s_0, …, s_{i-1}.
  # That is, ***before*** index i, PSL contains all permutations formed from elts s_0, …, s_{i-1}
  # And ***after*** index i, PSL contains all permutations formed from elts s_0, …, s_i



Using Loop Invariants in Algorithm Design and Explanations

- Loop invariants state properties known to be true for each iteration of a loop
  - Can think of it as a property known to be true immediately before and immediately after each iteration
- In our examples, we've understood loop invariants in terms of *before* and *after* conditions
  - *Before*: What is helpful and known to be true before a given iteration of the loop
  - *After*: What is helpful and known to be true after a given iteration of the loop, which establishes the truth of the *Before* condition before the next iteration
- You don't need to write down before / after conditions (unless a CS375 assignment requires it!), but it can be helpful
- Key point: Loop invariant gives a property that's true *after the loop is finished* that helps explain algo correctness

Example—let's use the invariant to help explain the correctness of our generate permutations algo:



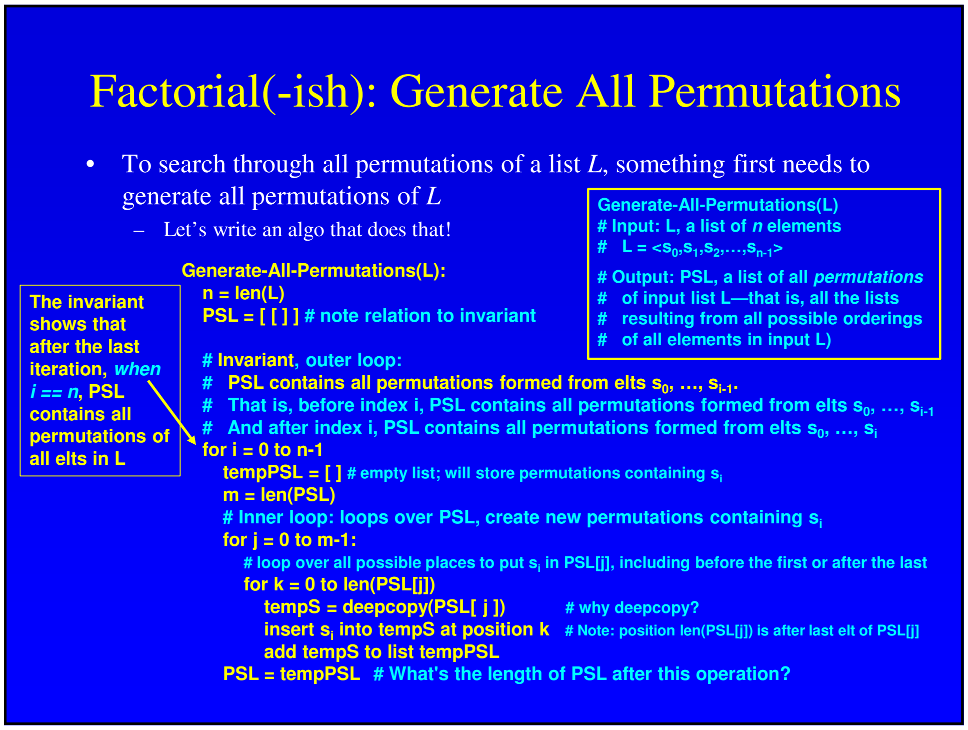
Factorial(-ish): Generate All Permutations

- To search through all permutations of a list *L*, something first needs to generate all permutations of *L*
  - Let's write an algo that does that!

```
Generate-All-Permutations(L)
# Input: L, a list of n elements
#   L = <$s_0, s_1, s_2, \dots, s_{n-1}$>
# Output: PSL, a list of all permutations
#   of input list L—that is, all the lists
#   resulting from all possible orderings
#   of all elements in input L)
```

How does the invariant help explain correctness—that the algo meets its specifications?

```
Generate-All-Permutations(L):
  n = len(L)
  PSL = [ [ ] ] # note relation to invariant

  # Invariant, outer loop:
  #   PSL contains all permutations formed from elts $s_0, \dots, s_{i-1}$.
  #   That is, before index i, PSL contains all permutations formed from elts $s_0, \dots, s_{i-1}$
  #   And after index i, PSL contains all permutations formed from elts $s_0, \dots, s_i$
  for i = 0 to n-1
    tempPSL = [ ] # empty list; will store permutations containing $s_i$
    m = len(PSL)
    # Inner loop: loops over PSL, create new permutations containing $s_i$
    for j = 0 to m-1:
      # loop over all possible places to put $s_i$ in PSL[j], including before the first or after the last
      for k = 0 to len(PSL[j])
        tempS = deepcopy(PSL[ j ])          # why deepcopy?
        insert $s_i$ into tempS at position k   # Note: position len(PSL[j]) is after last elt of PSL[j]
        add tempS to list tempPSL
    PSL = tempPSL   # What's the length of PSL after this operation?
```



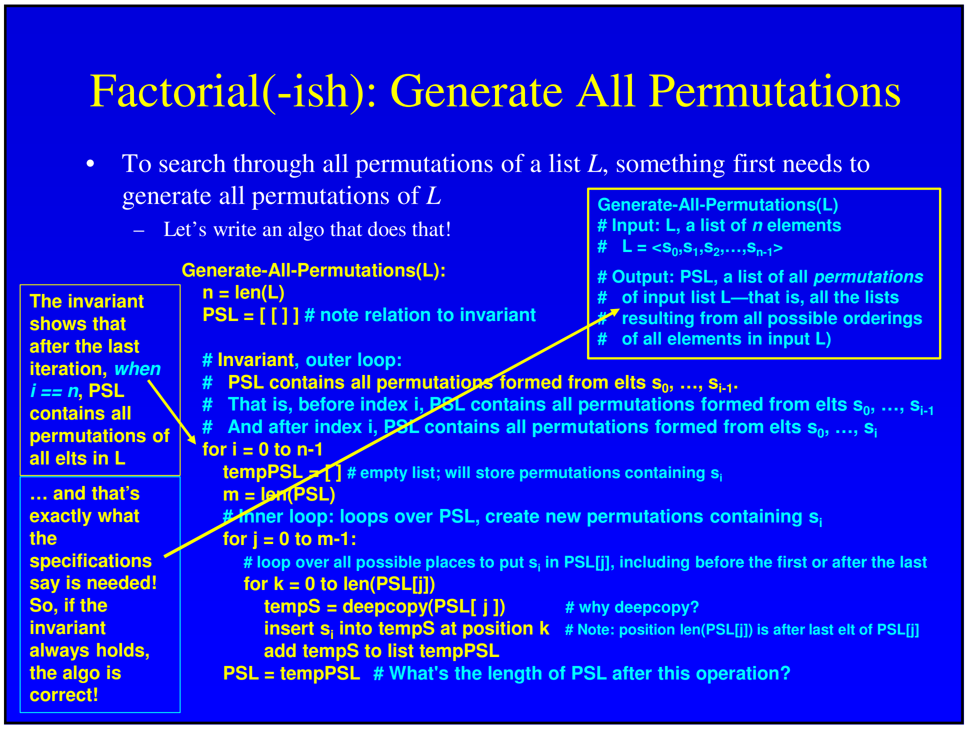
Factorial(-ish): Generate All Permutations

- To search through all permutations of a list *L*, something first needs to generate all permutations of *L*
  - Let's write an algo that does that!

Generate-All-Permutations(L)
# Input: L, a list of *n* elements
L = <s_0,s_1,s_2,…,s_{n-1}>
# Output: PSL, a list of all *permutations*
#   of input list L—that is, all the lists
#   resulting from all possible orderings
#   of all elements in input L)

The invariant shows that after the last iteration, *when i == n*, PSL contains all permutations of all elts in L

```
Generate-All-Permutations(L):
  n = len(L)
  PSL = [ [ ] ] # note relation to invariant

  # Invariant, outer loop:
  #   PSL contains all permutations formed from elts $s_0$, …, $s_{i-1}$.
  #   That is, before index i, PSL contains all permutations formed from elts $s_0$, …, $s_{i-1}$
  #   And after index i, PSL contains all permutations formed from elts $s_0$, …, $s_i$
  for i = 0 to n-1
    tempPSL = [ ] # empty list; will store permutations containing $s_i$
    m = len(PSL)
    # Inner loop: loops over PSL, create new permutations containing $s_i$
    for j = 0 to m-1:
      # loop over all possible places to put $s_i$ in PSL[j], including before the first or after the last
      for k = 0 to len(PSL[j])
        tempS = deepcopy(PSL[ j ])          # why deepcopy?
        insert $s_i$ into tempS at position k   # Note: position len(PSL[j]) is after last elt of PSL[j]
        add tempS to list tempPSL
    PSL = tempPSL  # What's the length of PSL after this operation?
```



Factorial(-ish): Generate All Permutations

- To search through all permutations of a list *L*, something first needs to generate all permutations of *L*
  - Let's write an algo that does that!

Generate-All-Permutations(L)
# Input: L, a list of *n* elements
L = <s_0,s_1,s_2,…,s_{n-1}>
# Output: PSL, a list of all *permutations*
#   of input list L—that is, all the lists
#   resulting from all possible orderings
#   of all elements in input L)

The invariant shows that after the last iteration, *when i == n*, PSL contains all permutations of all elts in L

… and that's exactly what the specifications say is needed! So, if the invariant always holds, the algo is correct!

```
Generate-All-Permutations(L):
  n = len(L)
  PSL = [ [ ] ] # note relation to invariant

  # Invariant, outer loop:
  #   PSL contains all permutations formed from elts $s_0$, …, $s_{i-1}$.
  #   That is, before index i, PSL contains all permutations formed from elts $s_0$, …, $s_{i-1}$
  #   And after index i, PSL contains all permutations formed from elts $s_0$, …, $s_i$
  for i = 0 to n-1
    tempPSL = [ ] # empty list; will store permutations containing $s_i$
    m = len(PSL)
    # Inner loop: loops over PSL, create new permutations containing $s_i$
    for j = 0 to m-1:
      # loop over all possible places to put $s_i$ in PSL[j], including before the first or after the last
      for k = 0 to len(PSL[j])
        tempS = deepcopy(PSL[ j ])          # why deepcopy?
        insert $s_i$ into tempS at position k   # Note: position len(PSL[j]) is after last elt of PSL[j]
        add tempS to list tempPSL
    PSL = tempPSL  # What's the length of PSL after this operation?
```

Exponential(-ish): Generate All Subsets

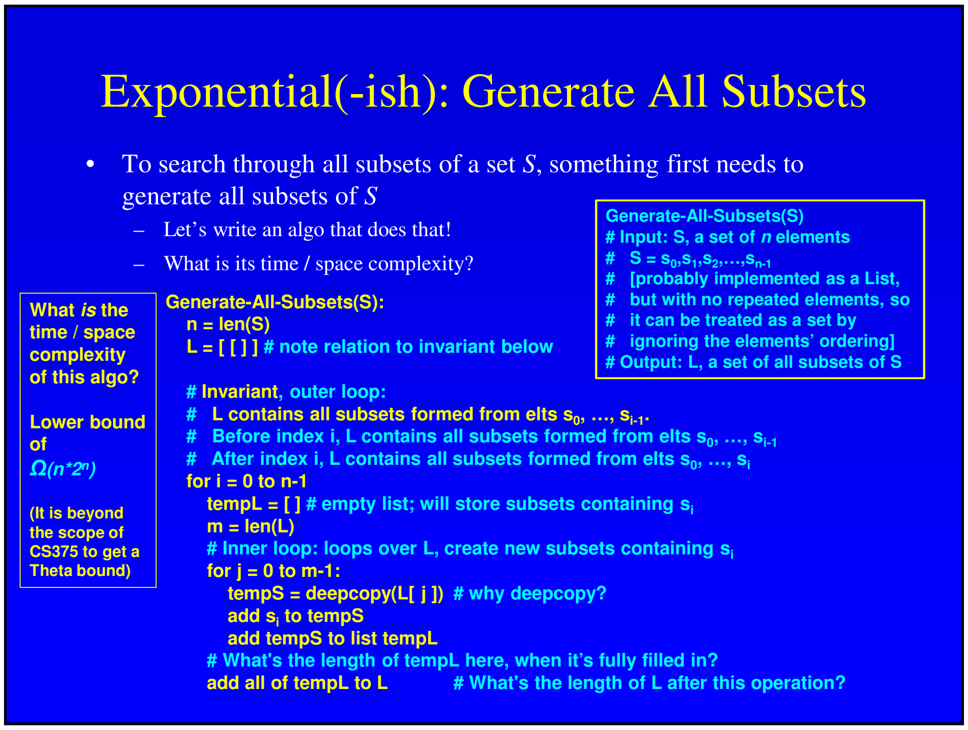
- To search through all subsets of a set *S*, something first needs to generate all subsets of *S*
  - Let’s write an algo that does that!
  - What is its time / space complexity?

```
Generate-All-Subsets(S)
# Input: S, a set of n elements
#   S = s_0,s_1,s_2,…,s_{n-1}
#   [probably implemented as a List,
#   but with no repeated elements, so
#   it can be treated as a set by
#   ignoring the elements’ ordering]
# Output: L, a set of all subsets of S
```

What *is* the time / space complexity of this algo?

What can you give as a lower bound (Ω bound)?

```
Generate-All-Subsets(S):
   n = len(S)
   L = [ [ ] ] # note relation to invariant below

   # Invariant, outer loop:
   #   L contains all subsets formed from elts s_0, …, s_{i-1}.
   #   Before index i, L contains all subsets formed from elts s_0, …, s_{i-1}
   #   After index i, L contains all subsets formed from elts s_0, …, s_i
   for i = 0 to n-1
      tempL = [ ] # empty list; will store subsets containing s_i
      m = len(L)
      # Inner loop: loops over L, create new subsets containing s_i
      for j = 0 to m-1:
         tempS = deepcopy(L[ j ])  # why deepcopy?
         add s_i to tempS
         add tempS to list tempL
      # What’s the length of tempL here, when it’s fully filled in?
      add all of tempL to L          # What’s the length of L after this operation?
```



Exponential(-ish): Generate All Subsets

- To search through all subsets of a set *S*, something first needs to generate all subsets of *S*
  - Let's write an algo that does that!
  - What is its time / space complexity?

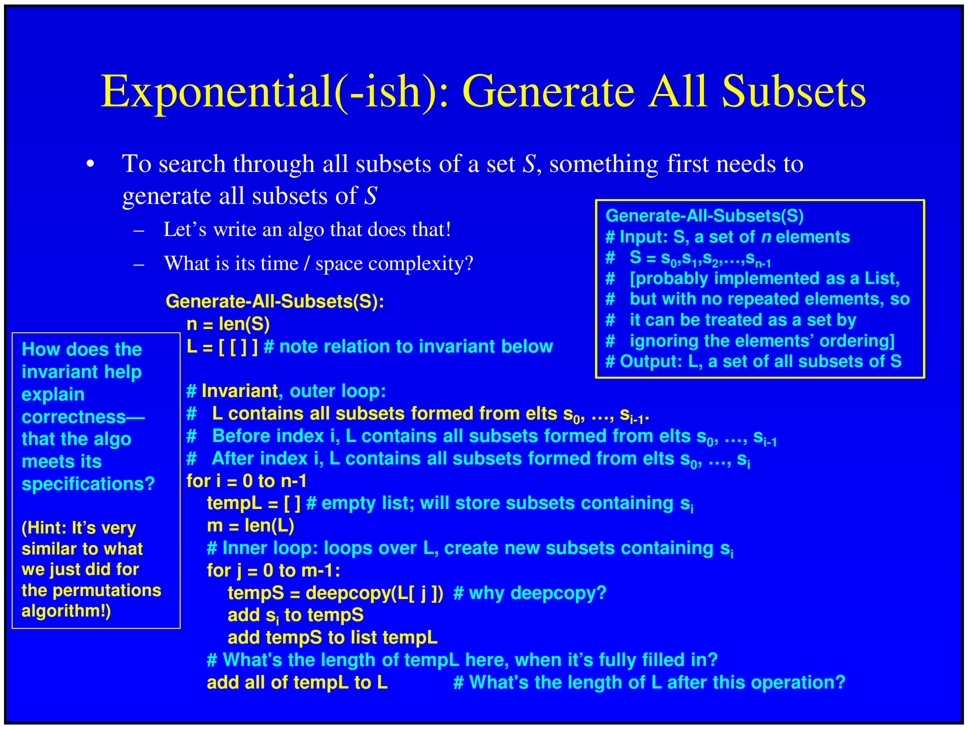
```
Generate-All-Subsets(S)
# Input: S, a set of n elements
#   S = s_0,s_1,s_2,…,s_{n-1}
#   [probably implemented as a List,
#   but with no repeated elements, so
#   it can be treated as a set by
#   ignoring the elements' ordering]
# Output: L, a set of all subsets of S
```

What *is* the time / space complexity of this algo?

Lower bound of *$\Omega(n*2^n)$*

(It is beyond the scope of CS375 to get a Theta bound)

```
Generate-All-Subsets(S):
   n = len(S)
   L = [ [ ] ] # note relation to invariant below

   # Invariant, outer loop:
   #   L contains all subsets formed from elts s_0, …, s_{i-1}.
   #   Before index i, L contains all subsets formed from elts s_0, …, s_{i-1}
   #   After index i, L contains all subsets formed from elts s_0, …, s_i
   for i = 0 to n-1
      tempL = [ ] # empty list; will store subsets containing s_i
      m = len(L)
      # Inner loop: loops over L, create new subsets containing s_i
      for j = 0 to m-1:
         tempS = deepcopy(L[ j ])  # why deepcopy?
         add s_i to tempS
         add tempS to list tempL
      # What's the length of tempL here, when it's fully filled in?
      add all of tempL to L          # What's the length of L after this operation?
```



Exponential(-ish): Generate All Subsets

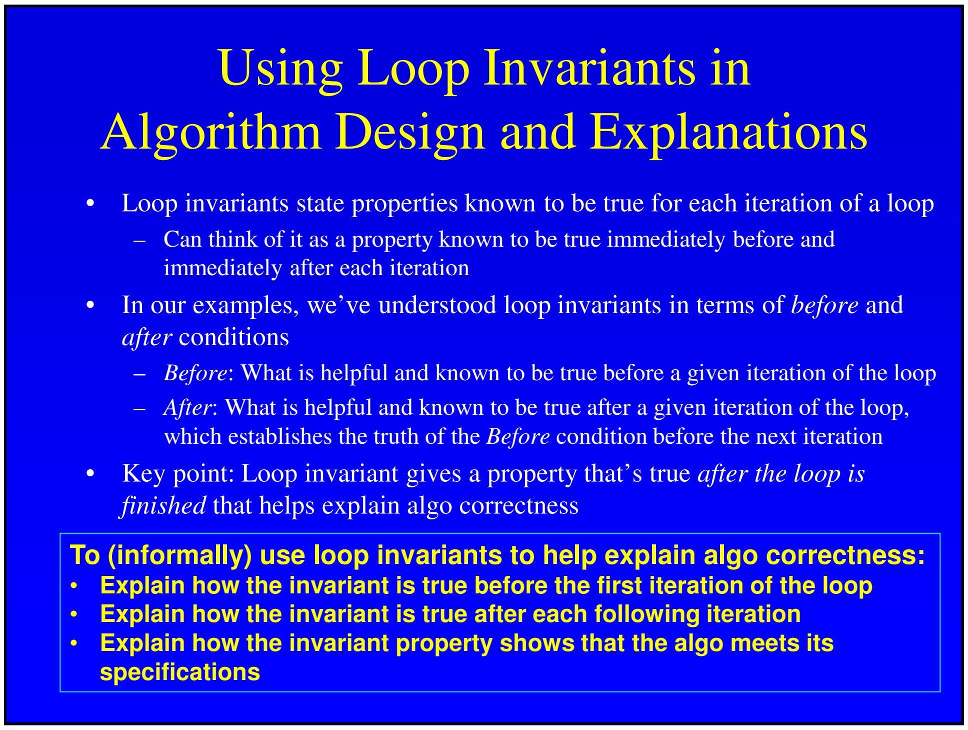
- To search through all subsets of a set *S*, something first needs to generate all subsets of *S*
  - Let's write an algo that does that!
  - What is its time / space complexity?

```
Generate-All-Subsets(S)
# Input: S, a set of n elements
#   S = s_0,s_1,s_2,…,s_{n-1}
#   [probably implemented as a List,
#   but with no repeated elements, so
#   it can be treated as a set by
#   ignoring the elements' ordering]
# Output: L, a set of all subsets of S
```

How does the invariant help explain correctness—that the algo meets its specifications?

(Hint: It's very similar to what we just did for the permutations algorithm!)

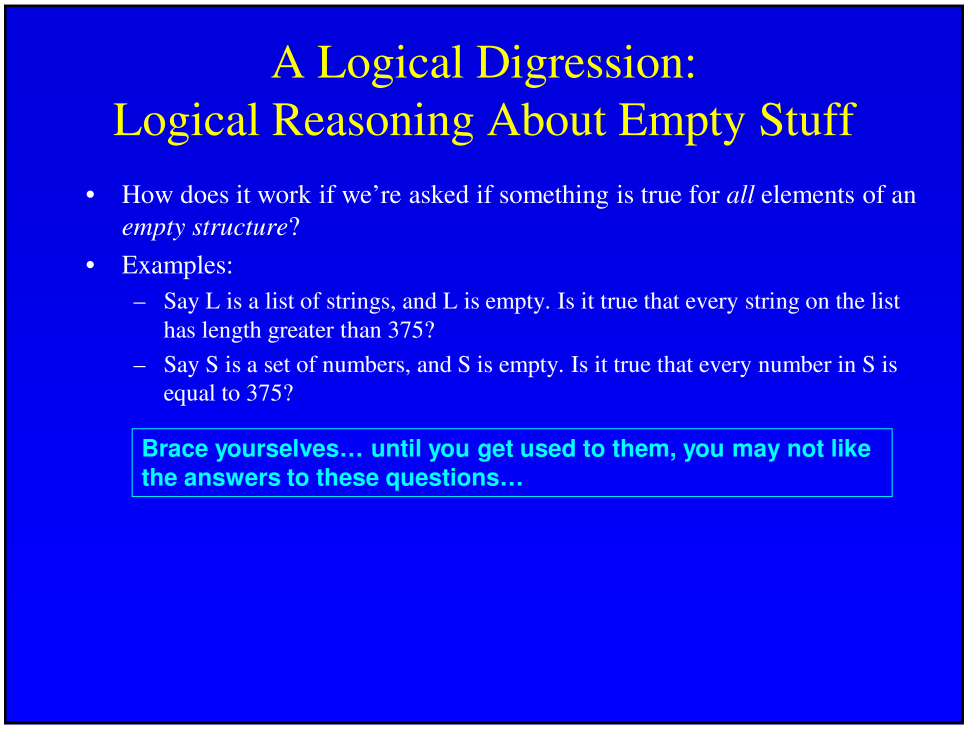
```
Generate-All-Subsets(S):
   n = len(S)
   L = [ [ ] ] # note relation to invariant below

   # Invariant, outer loop:
   #   L contains all subsets formed from elts s_0, …, s_{i-1}.
   #   Before index i, L contains all subsets formed from elts s_0, …, s_{i-1}
   #   After index i, L contains all subsets formed from elts s_0, …, s_i
   for i = 0 to n-1
      tempL = [ ] # empty list; will store subsets containing s_i
      m = len(L)
      # Inner loop: loops over L, create new subsets containing s_i
      for j = 0 to m-1:
         tempS = deepcopy(L[ j ])  # why deepcopy?
         add s_i to tempS
         add tempS to list tempL
      # What's the length of tempL here, when it's fully filled in?
      add all of tempL to L          # What's the length of L after this operation?
```



Using Loop Invariants in Algorithm Design and Explanations

- Loop invariants state properties known to be true for each iteration of a loop
  - Can think of it as a property known to be true immediately before and immediately after each iteration
- In our examples, we've understood loop invariants in terms of *before* and *after* conditions
  - *Before*: What is helpful and known to be true before a given iteration of the loop
  - *After*: What is helpful and known to be true after a given iteration of the loop, which establishes the truth of the *Before* condition before the next iteration
- Key point: Loop invariant gives a property that's true *after the loop is finished* that helps explain algo correctness

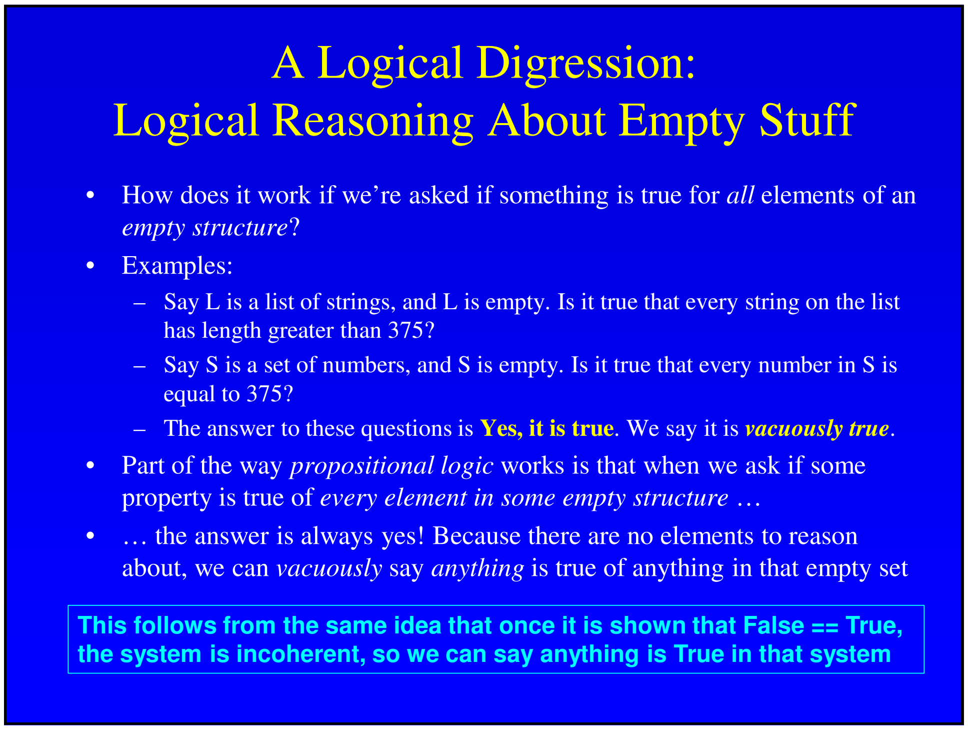
To (informally) use loop invariants to help explain algo correctness:
- **Explain how the invariant is true before the first iteration of the loop**
- **Explain how the invariant is true after each following iteration**
- **Explain how the invariant property shows that the algo meets its specifications**



A Logical Digression: Logical Reasoning About Empty Stuff

- How does it work if we're asked if something is true for *all* elements of an *empty structure*?
- Examples:
  - Say L is a list of strings, and L is empty. Is it true that every string on the list has length greater than 375?
  - Say S is a set of numbers, and S is empty. Is it true that every number in S is equal to 375?

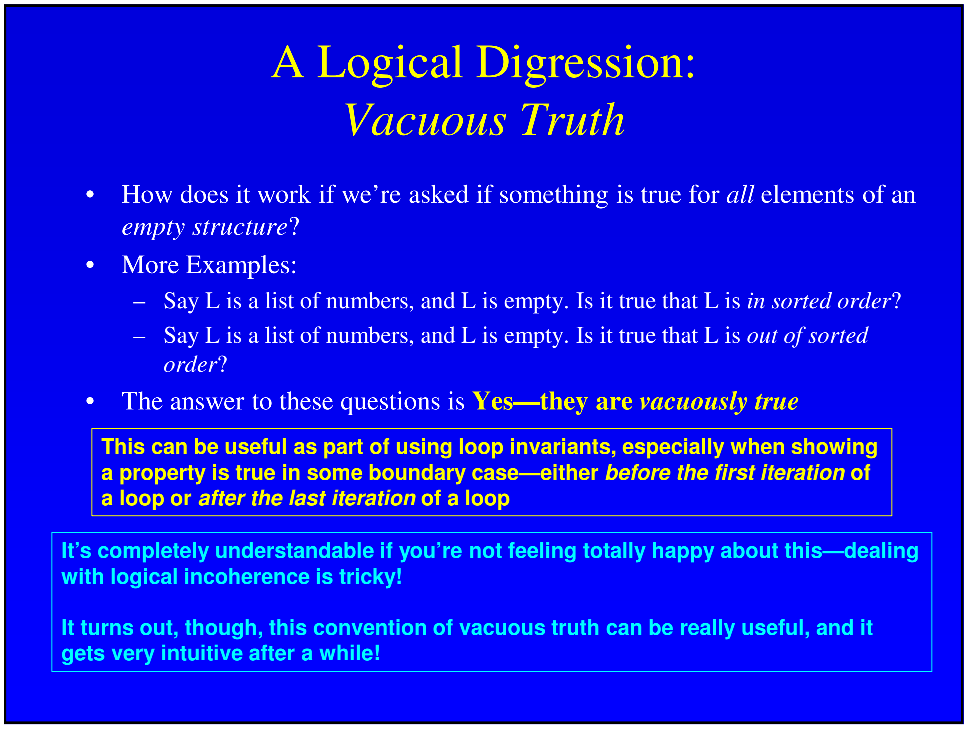
Brace yourselves… until you get used to them, you may not like the answers to these questions…



A Logical Digression: Logical Reasoning About Empty Stuff

- How does it work if we're asked if something is true for *all* elements of an *empty structure*?
- Examples:
  - Say L is a list of strings, and L is empty. Is it true that every string on the list has length greater than 375?
  - Say S is a set of numbers, and S is empty. Is it true that every number in S is equal to 375?
  - The answer to these questions is **Yes, it is true**. We say it is ***vacuously true***.
- Part of the way *propositional logic* works is that when we ask if some property is true of *every element in some empty structure …*
- … the answer is always yes! Because there are no elements to reason about, we can *vacuously* say *anything* is true of anything in that empty set

This follows from the same idea that once it is shown that False == True, the system is incoherent, so we can say anything is True in that system



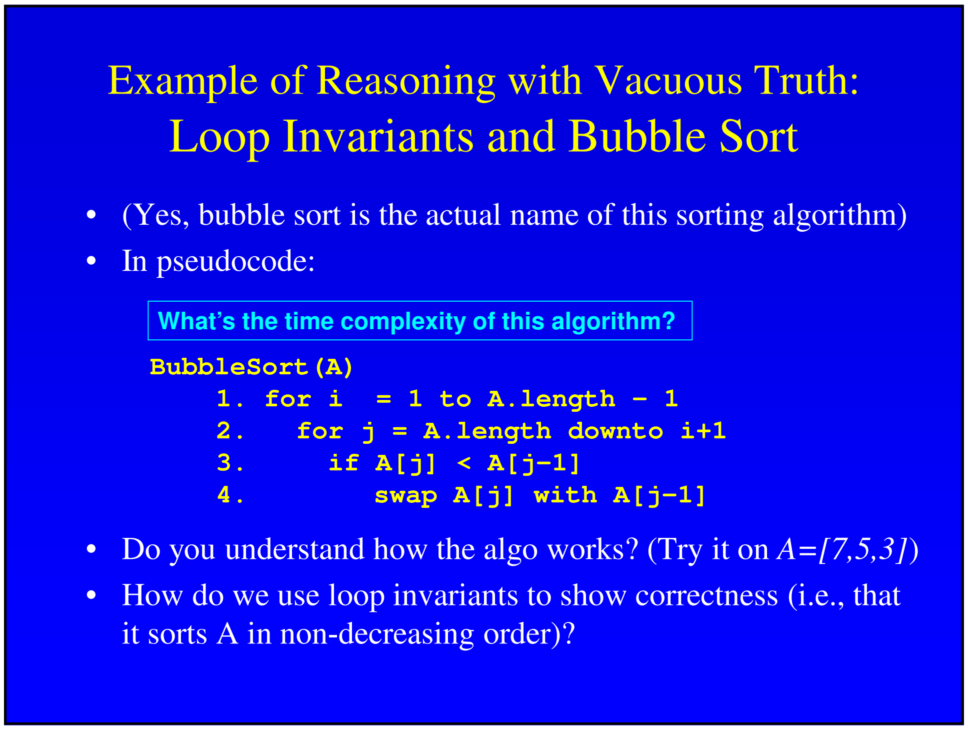
A Logical Digression: *Vacuous Truth*

- How does it work if we're asked if something is true for *all* elements of an *empty structure*?
- More Examples:
  - Say L is a list of numbers, and L is empty. Is it true that L is *in sorted order*?
  - Say L is a list of numbers, and L is empty. Is it true that L is *out of sorted order*?
- The answer to these questions is **Yes—they are *vacuously true***

This can be useful as part of using loop invariants, especially when showing a property is true in some boundary case—either *before the first iteration* of a loop or *after the last iteration* of a loop

It's completely understandable if you're not feeling totally happy about this—dealing with logical incoherence is tricky!

It turns out, though, this convention of vacuous truth can be really useful, and it gets very intuitive after a while!



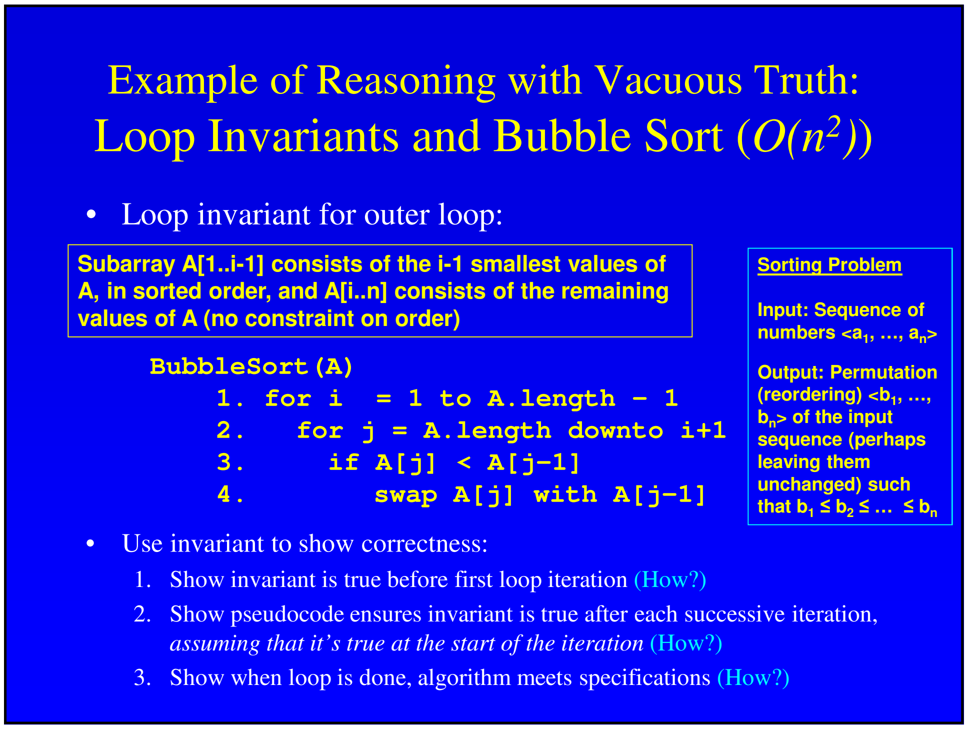
Example of Reasoning with Vacuous Truth: Loop Invariants and Bubble Sort

- (Yes, bubble sort is the actual name of this sorting algorithm)
- In pseudocode:

What's the time complexity of this algorithm?

```
BubbleSort(A)
    for i  = 1 to A.length - 1
      for j = A.length downto i+1
        if A[j] < A[j-1]
           swap A[j] with A[j-1]
```

- Do you understand how the algo works? (Try it on *A=[7,5,3]*)
- How do we use loop invariants to show correctness (i.e., that it sorts A in non-decreasing order)?



Example of Reasoning with Vacuous Truth: Loop Invariants and Bubble Sort ($O(n^2)$)

- Loop invariant for outer loop:

Subarray A[1..i-1] consists of the i-1 smallest values of A, in sorted order, and A[i..n] consists of the remaining values of A (no constraint on order)

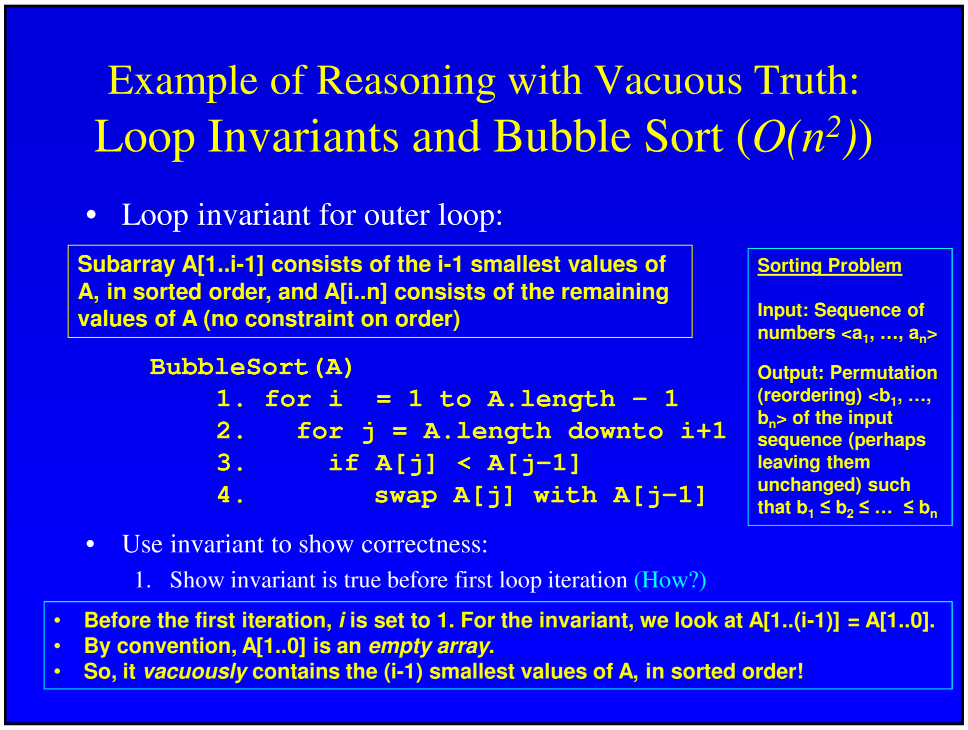
```
BubbleSort(A)
    for i  = 1 to A.length - 1
      for j = A.length downto i+1
        if A[j] < A[j-1]
           swap A[j] with A[j-1]
```

Sorting Problem

Input: Sequence of numbers $<a_1, \ldots, a_n>$

Output: Permutation (reordering) $<b_1, \ldots, b_n>$ of the input sequence (perhaps leaving them unchanged) such that $b_1 \leq b_2 \leq \ldots \leq b_n$

- Use invariant to show correctness:
  1. Show invariant is true before first loop iteration (How?)
  2. Show pseudocode ensures invariant is true after each successive iteration, *assuming that it's true at the start of the iteration* (How?)
  3. Show when loop is done, algorithm meets specifications (How?)



Example of Reasoning with Vacuous Truth: Loop Invariants and Bubble Sort ($O(n^2)$)

- Loop invariant for outer loop:

Subarray A[1..i-1] consists of the i-1 smallest values of A, in sorted order, and A[i..n] consists of the remaining values of A (no constraint on order)

```
BubbleSort(A)
    1. for i  = 1 to A.length – 1
    2.   for j = A.length downto i+1
    3.     if A[j] < A[j-1]
    4.        swap A[j] with A[j-1]
```

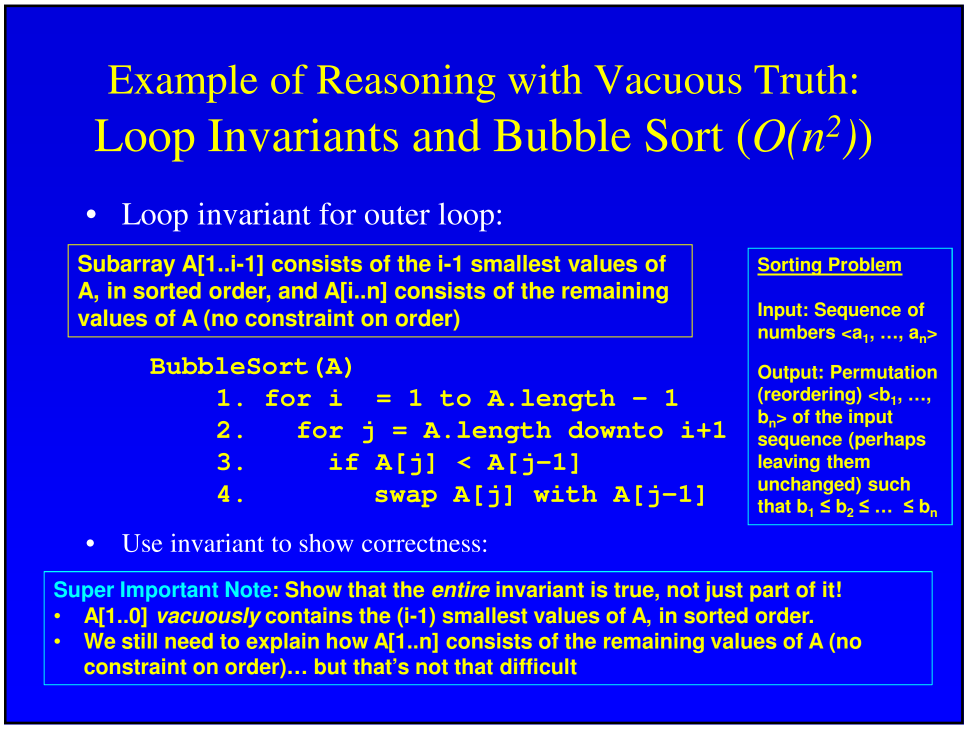
Sorting Problem

Input: Sequence of numbers $<a_1, \ldots, a_n>$

Output: Permutation (reordering) $<b_1, \ldots, b_n>$ of the input sequence (perhaps leaving them unchanged) such that $b_1 \leq b_2 \leq \ldots \leq b_n$

- Use invariant to show correctness:
  1. Show invariant is true before first loop iteration (How?)

- **Before the first iteration, *i* is set to 1. For the invariant, we look at A[1..(i-1)] = A[1..0].**
- **By convention, A[1..0] is an *empty array*.**
- **So, it *vacuously* contains the (i-1) smallest values of A, in sorted order!**



Example of Reasoning with Vacuous Truth: Loop Invariants and Bubble Sort ($O(n^2)$)

- Loop invariant for outer loop:

Subarray A[1..i-1] consists of the i-1 smallest values of A, in sorted order, and A[i..n] consists of the remaining values of A (no constraint on order)

```
BubbleSort(A)
    1. for i  = 1 to A.length – 1
    2.   for j = A.length downto i+1
    3.     if A[j] < A[j-1]
    4.        swap A[j] with A[j-1]
```

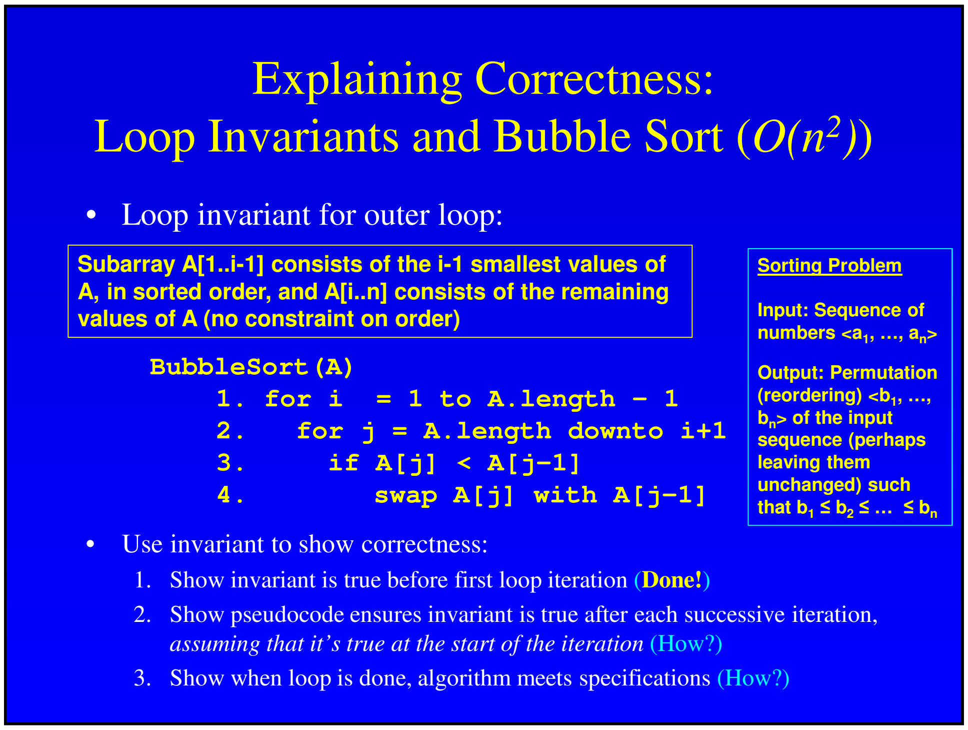
Sorting Problem

Input: Sequence of numbers $<a_1, \ldots, a_n>$

Output: Permutation (reordering) $<b_1, \ldots, b_n>$ of the input sequence (perhaps leaving them unchanged) such that $b_1 \leq b_2 \leq \ldots \leq b_n$

- Use invariant to show correctness:

Super Important Note: Show that the *entire* invariant is true, not just part of it!
- **A[1..0] *vacuously* contains the (i-1) smallest values of A, in sorted order.**
- **We still need to explain how A[1..n] consists of the remaining values of A (no constraint on order)… but that's not that difficult**



Explaining Correctness: Loop Invariants and Bubble Sort ($O(n^2)$)

- Loop invariant for outer loop:

Subarray A[1..i-1] consists of the i-1 smallest values of A, in sorted order, and A[i..n] consists of the remaining values of A (no constraint on order)

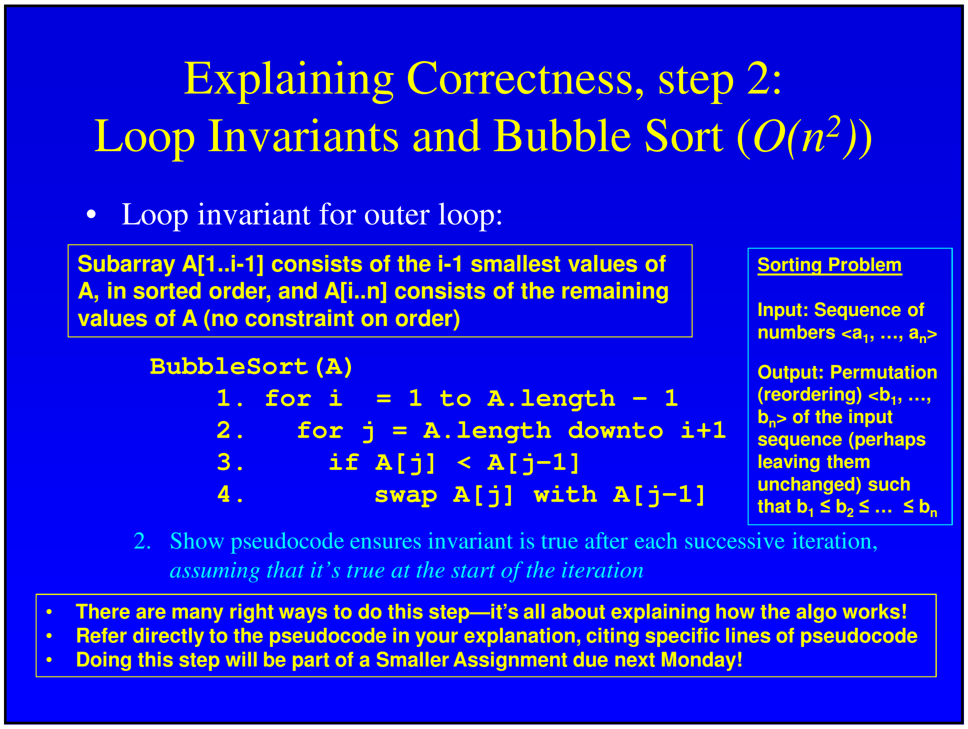
```
BubbleSort(A)
    1. for i  = 1 to A.length - 1
    2.   for j = A.length downto i+1
    3.     if A[j] < A[j-1]
    4.        swap A[j] with A[j-1]
```

Sorting Problem

Input: Sequence of numbers <a_1, …, a_n>

Output: Permutation (reordering) <b_1, …, b_n> of the input sequence (perhaps leaving them unchanged) such that $b_1 \leq b_2 \leq \ldots \leq b_n$

- Use invariant to show correctness:
  1. Show invariant is true before first loop iteration (**Done!**)
  2. Show pseudocode ensures invariant is true after each successive iteration, *assuming that it's true at the start of the iteration* (How?)
  3. Show when loop is done, algorithm meets specifications (How?)



Explaining Correctness, step 2: Loop Invariants and Bubble Sort ($O(n^2)$)

- Loop invariant for outer loop:

Subarray A[1..i-1] consists of the i-1 smallest values of A, in sorted order, and A[i..n] consists of the remaining values of A (no constraint on order)

```
BubbleSort(A)
    1. for i  = 1 to A.length - 1
    2.   for j = A.length downto i+1
    3.     if A[j] < A[j-1]
    4.        swap A[j] with A[j-1]
```

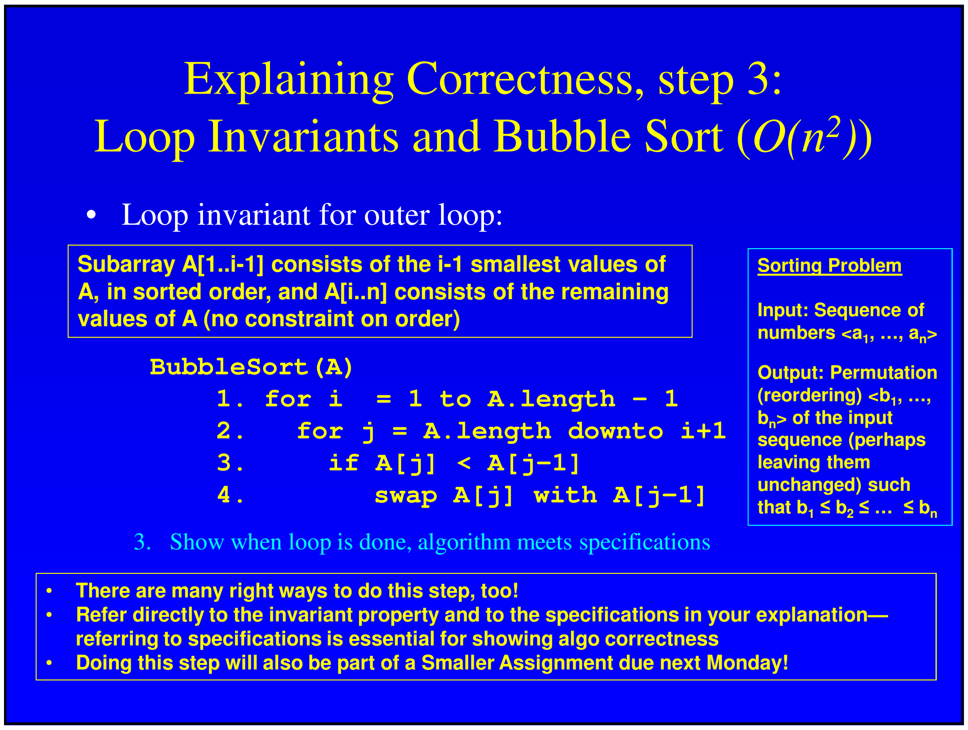
Sorting Problem

Input: Sequence of numbers <a_1, …, a_n>

Output: Permutation (reordering) <b_1, …, b_n> of the input sequence (perhaps leaving them unchanged) such that $b_1 \leq b_2 \leq \ldots \leq b_n$

2. Show pseudocode ensures invariant is true after each successive iteration, *assuming that it's true at the start of the iteration*

- **There are many right ways to do this step—it's all about explaining how the algo works!**
- **Refer directly to the pseudocode in your explanation, citing specific lines of pseudocode**
- **Doing this step will be part of a Smaller Assignment due next Monday!**



Explaining Correctness, step 3: Loop Invariants and Bubble Sort ($O(n^2)$)

- Loop invariant for outer loop:

Subarray A[1..i-1] consists of the i-1 smallest values of A, in sorted order, and A[i..n] consists of the remaining values of A (no constraint on order)

```
BubbleSort(A)
    1. for i  = 1 to A.length – 1
    2.   for j = A.length downto i+1
    3.     if A[j] < A[j-1]
    4.        swap A[j] with A[j-1]
```

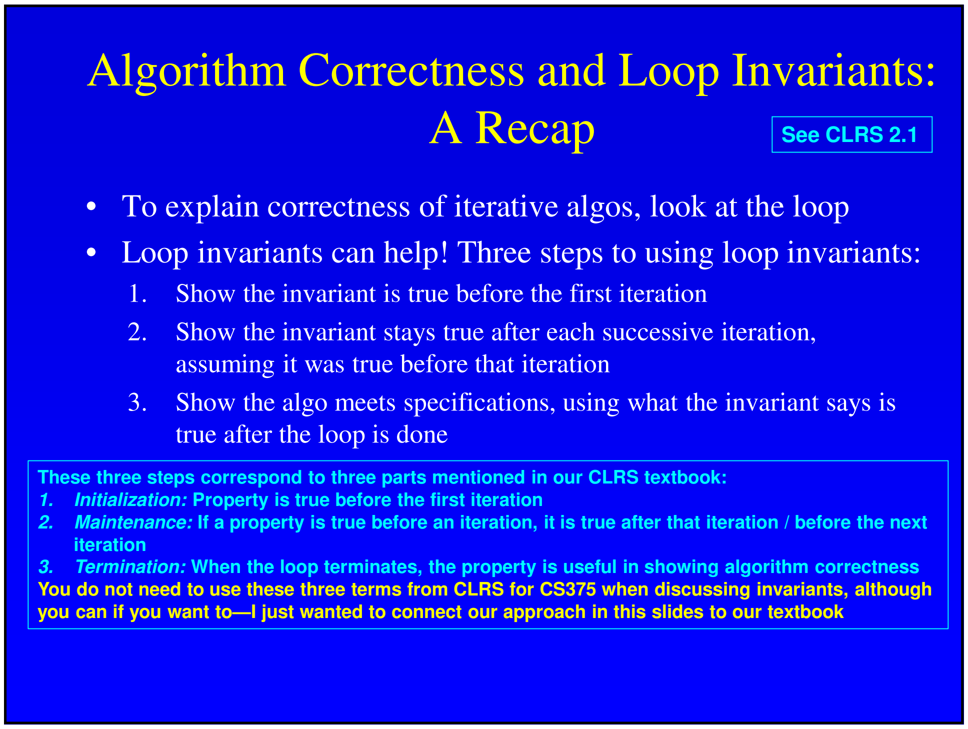
Sorting Problem

Input: Sequence of numbers $<a_1, \ldots, a_n>$

Output: Permutation (reordering) $<b_1, \ldots, b_n>$ of the input sequence (perhaps leaving them unchanged) such that $b_1 \leq b_2 \leq \ldots \leq b_n$

3. Show when loop is done, algorithm meets specifications

- **There are many right ways to do this step, too!**
- **Refer directly to the invariant property and to the specifications in your explanation—referring to specifications is essential for showing algo correctness**
- **Doing this step will also be part of a Smaller Assignment due next Monday!**



Algorithm Correctness and Loop Invariants: A Recap

See CLRS 2.1

- To explain correctness of iterative algos, look at the loop
- Loop invariants can help! Three steps to using loop invariants:
  1. Show the invariant is true before the first iteration
  2. Show the invariant stays true after each successive iteration, assuming it was true before that iteration
  3. Show the algo meets specifications, using what the invariant says is true after the loop is done

These three steps correspond to three parts mentioned in our CLRS textbook:
1. ***Initialization:* Property is true before the first iteration**
2. ***Maintenance:* If a property is true before an iteration, it is true after that iteration / before the next iteration**
3. ***Termination:* When the loop terminates, the property is useful in showing algorithm correctness**

You do not need to use these three terms from CLRS for CS375 when discussing invariants, although you can if you want to—I just wanted to connect our approach in this slides to our textbook