Introduction to Computational and Algorithmic Thinking

LECTURE 4 – SEARCHING AND SORTING ALGORITHMS. SCALABILITY

Announcements

This lecture: Searching and Sorting Algorithms. Scalability

Reading: Read Chapter 4 of Conery

Acknowledgement: Some of the lecture slides are based on CSE 101 lecture notes by Prof. Kevin McDonald at SBU and the textbook by John Conery.

Searching and sorting

- **Searching** is a common operation in many different situations:
  - Finding a file on your computer (e.g., Spotlight on MacOS, Search box in Windows Explorer)
  - Online dictionaries and catalogs
  - The “find” command in a word processor or text editor
  - Looking for a book, either on a bookshelf at home or in a library
  - Finding a name in a phone book or a word in a dictionary
  - Searching a file drawer to find customer information or student records

Searching and sorting

- What these searching problems have in common:
  - We have a potentially large collection of items
  - We need to search the collection to find a single item that matches a certain condition (e.g., name of book/name of a person)
- **Sorting** involves reorganizing information so it’s in a particular order
  - Sorting could help for faster searching
- There are many algorithms available for both searching and sorting of data

Iterative algorithms

- In this Unit we will look at one algorithm each for searching and sorting, and explore more later
- Both algorithms are *iterative* and rely heavily on loops
  - Searching algorithm: **linear search**
  - Sorting algorithm: **insertion sort**
- We will use the IterationLab module to help us explore these two algorithms

Linear search

- Linear search is the simplest, most straightforward search strategy
- As the name implies, the idea is to start at the beginning of a collection and compare items one after another
  - Also called **sequential search** because the elements of the collection are examined sequentially
- Recall the **index** method for lists, which tells us the position of an element in a list
- Example:

```
notes = ['do', 're', 'me', 'fa', 'sol', 'la', 'ti’]
notes.index('sol’) # will return the value 4
```

- The **index** method is performing a search, in fact

Linear search

- Also recall the **in** operator, which tells us if an element is present in a list
- This operator is necessary because the **index** method will cause our program to crash if the element we want is missing

```
if 'sol' in notes:
    print('Present in list...')
else:
    print('Not present in list...’)
```

Linear search

- The linear search function in IterationLab, **isearch**, is like Python's **index** method
  - Pass it a list and an item to search for
  - If the item is in the list, the function returns the location where it was found
  - If the item is not in the list, the function returns **None**
- Examples:

```
from PythonLabs.IterationLab import isearch
notes = ['do', 're', 'me', 'fa', 'sol', 'la', 'ti']
print(isearch(notes, 'ti')) # returns 6
print(isearch(notes, 'ba')) # returns None
```

- See isearch_tests.py if you installed PythonLabs

PythonLabs: RandomList

- For our experiments on searching and sorting algorithms we're going to need some data to test our programs
- The lab module defines a special type of list called a RandomList
  - We can make lists of integers or strings
  - The list will not contain any duplicates
  - When we do a searching experiment, we can use items we know are in the list (so the search succeeds) or not in the list (so the search fails)

RandomList examples

- List of 10 random integers:

```
from PythonLabs.Tools import RandomList
rand_nums = PythonLabs.Tools.RandomList(10)
print(rand_nums)
```

- Sample output:

```
[84, 62, 76, 24, 80, 42, 17, 54, 7, 14]
```

- List of 5 random fish (!!!):

```
fish = PythonLabs.Tools.RandomList(5, 'fish’)
print(fish)
```

- Sample output:

```
['black bass', 'halibut', 'herring’,
'flounder', 'mackerel’]
```

RandomList examples

- After we make a RandomList object, we can ask it to give us a randomly chosen item from the list
- First let's search for a fish that *is* in the list:

```
success_fish = fish.random('success')
isearch(fish, success_fish) # returns 4
```

- Now let's request the name of a fish that is not in the list:

```
fail_fish = fish.random('fail')
isearch(fish, fail_fish) # returns None
```

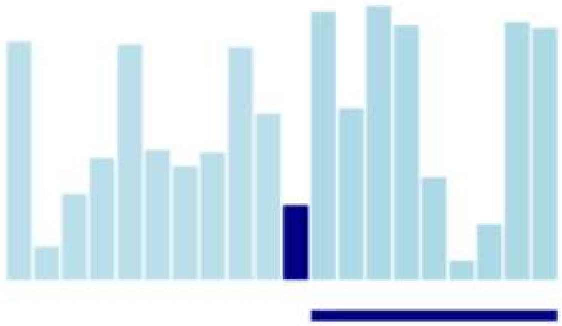
- See isearch_tests.py

Visualizing isearch()

- IterationLab supports visualization of **isearch**
- Visualizing the action will help us design and implement the algorithm
- First, make a list of random integers
- Next, print the list in the terminal window
- Then, pick a random number to search for
- Next, display the list on the canvas (works only for integers)
- Finally, call the **isearch** function on the list and display the result
- The Python code for this algorithm is given on the next slide and in isearch_visualization.py

Visualizing isearch()

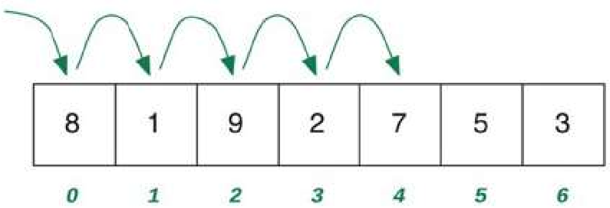
```
from PythonLabs.IterationLab import view_list, isearch
from PythonLabs.Tools import RandomList

nums = RandomList(20)
print('nums: ' + str(nums))
target = nums.random('success')
print('target: ' + str(target))
view_list(nums)
result = isearch(nums, target)
print('result: ' + str(result))
```



Implementing linear search

- One way to write our own version of **isearch** is to use a for-loop with a **range** expression

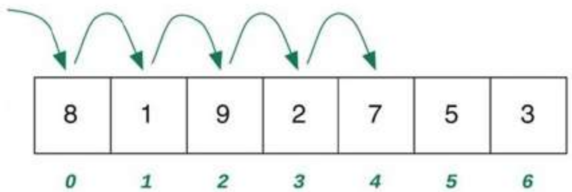
```
def isearch(a, x):
    for i in range(len(a)):
        if a[i] == x:
            return i
    return None
```



- Some explanation: the **i** variable acts as an index into **a**
- We do this so that we can return the position (index) of the target element, **x**, in the list

Implementing linear search

```
def isearch(a, x):
    for i in range(len(a)):
        if a[i] == x:
            return i
    return None
```

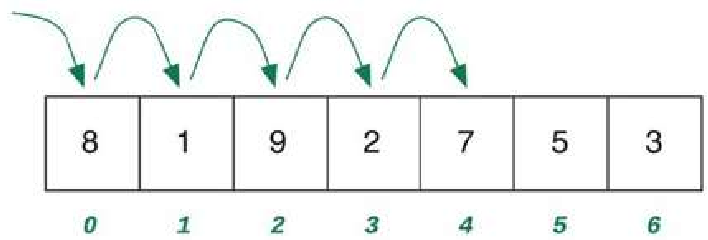


- If the item is found, the first return statement tells Python to exit the loop and return before the iteration is done
- If the item is not in the list, the loop terminates and the other return statement is executed returning **None**

while-loops

- Another way to write the function is shown below

```
def isearch(a, x):
    i = 0
    while i < len(a):
        if a[i] == x:
            return i
        i = i + 1
    return None
```



- The **while** statement is another kind of loop available in Python that is just as important as the **for** statement

while-loops

- Python evaluates the Boolean expression next to the keyword **while**
- If the expression is true, the statements in the body of the loop are executed
- Python then goes back to the top of the loop to evaluate the Boolean expression again

```
def isearch(a, x):
    i = 0
    while i < len(a):
        if a[i] == x:
            return i
        i = i + 1
    return None
```

while-loops

- The loop terminates when the Boolean expression becomes false

- Note that the index variable **I** must be initialized before the while statement

- The **i** variable is updated inside of the loop

- If **i** never changes, the program will be caught in an **infinite loop**

```
def isearch(a, x):
    i = 0
    while i < len(a):
        if a[i] == x:
            return i
        i = i + 1
    return None
```

while-loops vs. for-loops

- Why does Python give us two ways to write loops?
- for-loops are convenient, but some algorithms need to look at items in a different order
- for-loops are appropriate when you know or can calculate the number of times the loop's body must be executed
    - Sometimes we can't determine ahead of time the number of repetitions, so we use a while-loop instead
- So, for-loops are perfect for visiting every element of a list, and while-loops are the better choice for most other situations

while-loops vs. for-loops

- Advice: unless there is a good reason to use a while-loop, write your loops with for statements
  - Python takes care of initializing and updating the index variable
  - Programs will be shorter, simpler and less likely to contain errors
- The single line of code **for i in range(len(a))** would require three lines if written as a while-loop:

```
i = 0
while i < len(a):
    ...
    i = i + 1
```

Linear search: performance

- The linear search algorithm looks for an item in a list
  - Start at the beginning (**a[0]**, or “the left”)
  - Compare each item, moving systematically to the right (**i = i + 1**)
- How many comparisons will the linear search algorithm make as it searches through a list with *n* items?
  - Another way to phrase it: how many iterations will our Python function make in its while-loop?
  - Well, it depends on whether the search is successful or not, doesn’t it?!

Linear search: performance

- For an unsuccessful search:
  - Visit every item before returning None
  - i.e., make n comparisons
- For a successful search, anywhere from 1 and n interactions are required
  - Search may be lucky and find the item in the first location
  - At the other extreme, the item might be in the last location
  - Expect, on average, n/2 comparisons

Example: the Luhn Algorithm

- The Luhn algorithm checks if an account number (such as a credit card number) is valid

- It works like this:

1. Process each digit in turn, from *right to left*. The rightmost digit is treated as being in position #1.
   - Odd-positioned digits are added as-is to a running total.
   - Even-positioned digits are doubled. If that doubled value is less than 10, add it to the running total. Otherwise, add the two digits individually to the running total.
2. If the sum is a multiple of 10, the account number is valid. Otherwise it isn't.

Example: the Luhn Algorithm

- Here's an example of this computation
- Account number 79927398713
- Odd-positioned values: 3, 7, 9, 7, 9, 7 (remember: indexes start from 1 for this algorithm)
- Add them: 3+7+9+7+9+7=42
- Even-positioned values: 1, 8, 3, 2, 9
- Even-positioned values doubled: 2, 16, 6, 4, 18
- 16 and 18 are both > 10, so we will add 7 (1 + 6) and 9 (1 + 8) to the total
- Add: 42+2+7+6+4+9=70
- 70 is divisible by 10, so the account number is valid

Example: the Luhn Algorithm

- We know how to tell if a number is divisible by 10, right?

- The remainder operator (%):

```
if num % 10 == 0: # say "num mod 10"
    # stmts for when num is divisible by 10
else:
    # stmts for when num is not div. by 10
```

- We also know how to tell if a number is even or odd, right?

Example: the Luhn Algorithm

Sometimes it is useful (or necessary) to put one if-statement inside of another if-statement

These are known as **nested if-statements** as we saw in our last PS

We will find nested if-statements useful in implementing the Luhn algorithm

Python source code for the algorithm is given in a few slides, but see the file luhn.py itself for fully-commented code that explains every line of the source code

Example: the Luhn Algorithm

- Some additional Python functionality that will be useful in implementing the Luhn algorithm:
  - We can write ***=** and **//=** to multiply or divide (respectively) one number by another
  - Example: **salary *= 3** would triple the value stored in variable **salary**
- To extract digits one-by-one from an integer, we can use repeated division by 10:
  - **num % 10** would give us the rightmost digit of 10
  - **num //= 10** would then remove that digit from **num**
  - Suppose **num** is 942. **num % 10** would give us 2
  - Then **num //= 10** would change **num** from 942 to 94

Example: luhn.py

```
def luhn(number):
   total = 0
   position = 1
   while number > 0:
      digit = number % 10
      if position % 2 == 1:
         total += digit
      else:
         digit *= 2
         if digit >= 10:
            total += 1 + (digit % 10)
         else:
            total += digit
      number //= 10
      position += 1
   return total % 10 == 0
```

Sorting

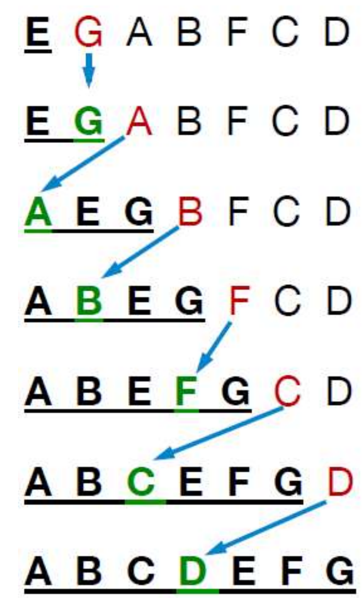
- The linear search algorithm is an example of an iterative algorithm
  - Start at the beginning of a collection
  - Systematically progress through the collection, all the way to the end, if necessary
- A similar strategy can be used to sort the items in a list
- We will now look at a simple, iterative sorting algorithm known as **insertion sort**
- The basic idea is:
  1. Pick up an item, find the place it belongs, insert it back into the list
  2. Move to the next item and repeat

Insertion sort

- The important property of the insertion sort algorithm: at any point in this algorithm, *part of the list is already sorted*
- More specifically, the left-hand part of the list is the sorted part and the right-hand part is still unsorted
    1. The initial item to work on is at index 1
    2. Pick up the current item
    3. Scan the left-hand part backwards from that index until we find an item lower than the current item or we arrive at the front of the list, whichever comes first
    4. Insert the current item back into the list at this location
    5. The next item to work on is to the right of the original location of the item
    6. Go back to step 2

Insertion sort example

- The example here illustrates the general idea
- The underlined letters constitute the sorted part of the array
- Initially, the leftmost item is considered to be in a sorted sub-list by itself, and all the other items are in an adjoining unsorted sub-list
- The leftmost item in the unsorted sub-list is selected and *inserted* into its correct position in the sorted sub-list



Insertion sort in Python (almost)

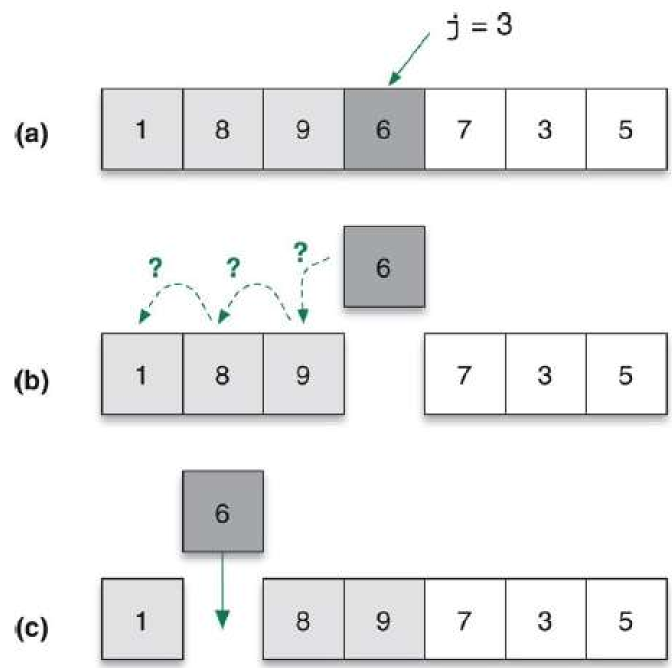
```
def isort(a):
  i = 1
  while i < len(a):
    x = a[i]
    remove x from a
    j = location for x
    insert x at a[j+1]
```

- We have a couple of obvious gaps in this pseudocode, but we're on our way to writing a Python function that implements insertion sort

Insertion sort in Python (almost)

- The figure on the next slide shows the core part of the algorithm:
  - Select the next item (step (a))
  - Remove the item from the list (step (b))
  - Determine at which index the item should go (step (b) also)
  - Insert the item at that index (step (c))
- When we're working on the item at index 3, the values to the left (indexes 0 through 2) have been sorted
- The statements in the body of the while-loop find the new location for this item and insert it back into the list

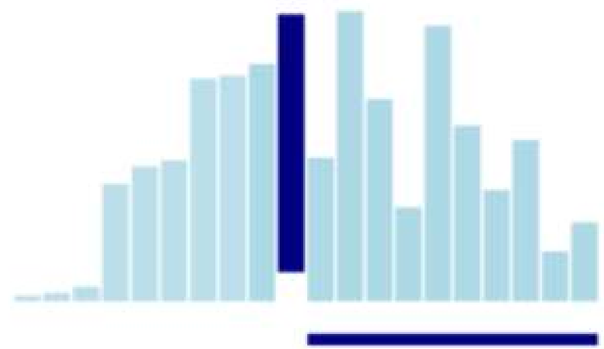
Insertion sort in Python (almost)



Visualizing inserting sort

- Before delving any deeper yet into the code, it might be helpful to watch a visualization of insertion sort in action
- See isort_visualization.py
- Also see http://visualgo.net/en/sorting

```
from PythonLabs.IterationLab
   import view_list, isort
from PythonLabs.Tools
   import RandomList
nums = RandomList(20)
print('nums before sorting: ' + str(nums))
view_list(nums)
isort(nums)
print('nums after sorting: ' + str(nums))
```



Moving items in a list

- In order to implement the steps in the body of the main loop we need to know:
  - How to remove an item from the middle of a list
  - How to insert an item into a list
- Both operations are performed by methods of the **`list`** class
- Call **`a.pop(i)`** to delete the item at location **`i`** in list **`a`**
  - The method returns the item that was deleted
- Call **`a.insert(i, x)`** to insert item **`x`** into the list **`a`** at location **`i`**
- Some examples of these methods are on the next slide

Moving items in a list

- Suppose we had a random list of seven chemical elements (e.g., oxygen, hydrogen, etc.):

  a = RandomList(7, 'elements’)

  a: ['Co', 'Tm', 'U', 'Hs', 'F', 'Rn', 'Y']

- Now let's remove the item at index 4 and save it in x:

  x = a.pop(4) # x will contain 'F'

- The list **a** will become:

  ['Co', 'Tm', 'U', 'Hs', 'Rn', 'Y']

- Let’s insert the element at index 2:

  a.insert(2, x)

- The list **a** will become:

  ['Co', 'Tm', 'F', 'U', 'Hs', 'Rn', 'Y’]

Finding where an item belongs

- During the insertion sort algorithm, after we pull an item out of the list, we have to find the location to re-insert it
- We'll use an index variable **j** to specify which locations we are checking
- Subtracting 1 from **j** will tell Python to move "left" during its search
- We can use a while-loop that keeps subtracting 1 from **j** until it finds the place where item **x** belongs

```
while a[j-1] > x:    # preliminary version
    j = j – 1        # of loop
```

Finding where an item belongs

- There is a potential problem with this strategy: what if **x** is smaller than everything to its left?
- The loop will reach a point where **j = 0** and there is nothing remaining to compare
- It will try to compare **x** to **a[-1]**, which is an error (index out of bound)
- The solution is to keep iterating only if **j > 0** *and* the item to the left is greater than **x**

```
while j > 0 and a[j-1] > x:    # final version
    j = j – 1                  # of loop
```

moveLeft()

- We can now write a helper function named **moveLeft** that inserts an item where it needs to go
- A call to **moveLeft(a, j)** will remove the item at **a[j]** and insert it back into **a** where it belongs

```
def moveLeft(a, j):
    x = a.pop(j)
    while j > 0 and a[j-1] > x:
        j -= 1
    a.insert(j, x)
```

- Example: let **a = [1, 3, 4, 6, 2, 7, 5]**
- After **moveLeft(a, 4)**, **a** is **[1, 2, 3, 4, 6, 7, 5]**

Completed isort() function

- With a helper function to move items, writing **isort** is easy
- Use a for-loop where an index variable **i** marks the start of the unsorted region
  - Initially **i** will be 1 (the single item at **a[0]** is a sorted region of size 1)
  - In the body of the loop, just call **moveLeft** to move the item at location **i** to its proper position
- **isort** is the top-level function called to solve the entire problem of sorting a list of numbers

```
def isort(a):
    for i in range(1, len(a)):
        moveLeft(a, i)
```

Completed isort() function

```
def isort(a):
   for i in range(1, len(a)):
      moveLeft(a, i)
```

- An example of how to use the **isort** function:

```
nums = RandomList(10)
isort(a)
# a is now sorted
```

- See also isort_visualization.py if you installed PythonLabs

Aside: Boolean operators

- In the process of implementing the **moveLeft** function we used a new Python keyword: **and**
- **and**, **or** and **not** are three of the Boolean operators that Python provides for writing Boolean conditions in if-statements and while-loops
- **p and q**: True only when Boolean variables **p** and **q** are both **True**
- **p or q**: True if either **p** or **q** (or both of them) is **True**
  - Note how this differs from the "or" used in everyday English
- **not p**: **True** if **p** is **False**; and **False** if **p** is **True**

Aside: Boolean operators

- Complex expressions can also have parentheses to form groups, as in **p and not (q or r)**
- Python performs a kind of “lazy evaluation”, meaning it evaluates a Boolean expression according to the rules of precedence and stops as soon as it can determine if the entire expression will be **True** or **False**
- For example, Python will evaluate the **j > 0** part of **while j > 0 and a[j-1] > x** before the **a[j-1] > x** part
- If it determines that **j > 0** is **False**, there is no need to evaluate **a[j-1] > x** because **False and "anything"** is always **False**

Example: Is it a leap year?

- Let's look at an example of Boolean expressions
- A year is a leap year if:
  - It is greater than 1582, and
  - It is divisible by 4, except centenary years not divisible by 400 (e.g., 1700, 1800, 1900, 2100, etc.)
- Here is one way of expressing this definition in code:
  - if the year is divisible by 4 and not 100, then it is a leap year
  - else, if the year is divisible by 400, then it is a leap year
  - otherwise, the year is not a leap year
- This logic is implemented in the code on the next slide, color-coded to map the algorithm to Python code

Example: leapyear.py

```
year = int(input('Enter a year: ‘))
if year < 1582:
   print('You must enter a year >= 1582.')
else:
   if ((year % 4 == 0) and (year % 100 != 0)) or
      (year % 400 == 0):
      print('That is a leap year.’)
   else:
      print('That is NOT a leap year.’)
```

- Example leap years: 2012, 2000, 2400
- Not leap years: 2003, 1900

Example: Crazy grading scheme

- Let's look at another example of Boolean expressions
- Students in Prof. Smith's math class take two regular exams and a final exam. The course grade is usually calculated as:

  (25% * exam #1 score) + (25% * exam #2 score)
  + (50% * final exam score)
- However, if a student scores less than a 60 on exam #1 or exam #2 (or both), he or she fails the course with a grade of 50, regardless of the other grades (Rule #1)
- Or, if a student scores 90 or higher on both exam #1 and exam #2, the course grade is the average of these two scores, provided that the normal weighted course average is *less than* the average of those two scores (Rule #2)

Example: Crazy grading scheme

- Here are some examples of how the rules apply
- exam #1: 48 exam #2: 92 final exam: 89
  - Rule #1 applies: the grade is 50
- exam #1: 92 exam #2: 91 final exam: 60
  - Rule #2 applies because the average of 92 and 91 is greater than the normal weighted grade
- exam #1: 92 exam #2: 91 final exam: 100
  - Don't apply Rule #2 because the normal weighted grade is higher than the average of 92 and 91
- exam #1: 62 exam #2: 90 final exam: 75
  - Neither one of the special rules applies, so we just take the normal weighted grade

Example: grades.py

```
def compute_grade(exam1_score, exam2_score,
                  final_exam_score):
  normal = 0.25 * exam1_score + 0.25 * exam2_score
           + 0.5 * final_exam_score

  if exam1_score < 60 or exam2_score < 60:
    grade = 50
  elif exam1_score >= 90 and
      exam2_score >= 90 and
      exam1_score + exam2_score) / 2 > normal:
    grade = (exam1_score + exam2_score) / 2
  else:
    grade = normal

  return grade
```

Rule #1

Rule #2

isort: Algorithm performance

- We saw earlier that for a list with n items we can expect, on average, to do n/2 comparisons during a linear search
- Can we come up with a similar equation for insertion sort?
- At first glance, it might seem that insertion sort is a "linear" algorithm like linear search
  - It has a for-loop that progresses through the list from left to right
- But remember that `moveLeft` also contains a loop
- The step that finds the proper location for the current item is also a loop
- It scans left from location `i`, going all the way back to 0 if necessary

isort: Algorithm performance

If we write **isort** without the **moveLeft** helper function, we can see that one loop is inside another

```
def isort(a):
   for i in range(1, len(a)):
      j = i
      x = a.pop(j)
      while j > 0 and a[j-1] > x:
         j = j - 1
      a.insert(j, x)
```

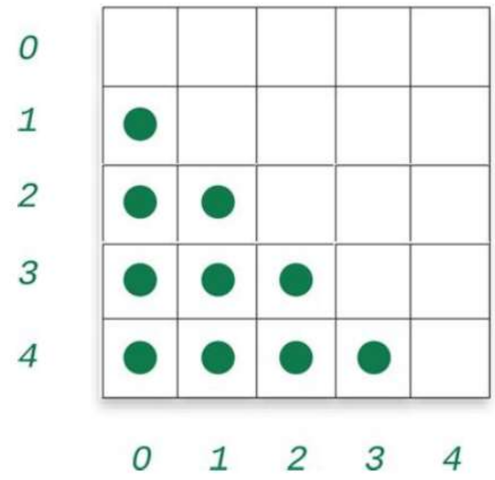
isort: Algorithm performance

- The outer loop has the same structure as the iteration in linear search
- A list index **i** ranges from 1 up to **n-1**
- At any time, the items to the left of **i** are sorted
- The inner loop moves **a[i]** to its proper location in the sorted region
- The size of the sorted region grows on each iteration
- We need to understand **nested loops** a bit better to be able to analyze this code

```
def isort(a):
  for i in range(1, len(a)):
    j = i
    x = a.pop(j)
    while j > 0 and a[j-1] > x:
      j = j - 1
    a.insert(j, x)
```

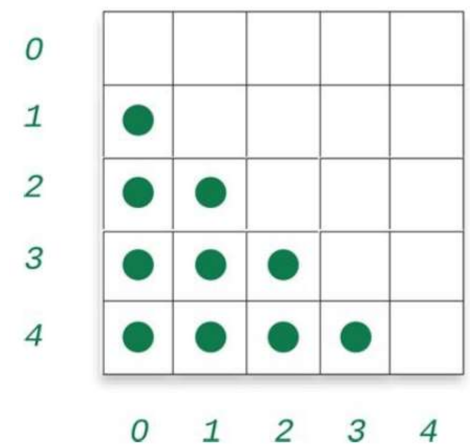
Nested loops

- An algorithm like **isort** that has one loop inside another is said to have **nested loops**
- In the figure below, a dot in a square indicates a potential comparison
- The row number is **i** from the code on the previous slide
- The column number is **j**
- For any value of **i**, the inner loop might have to compare values from **a[i-1]** all the way down to **a[0]**



Nested loops

```
def isort(a):
    for i in range(1, len(a)):
        j = i
        x = a.pop(j)
            while j > 0 and a[j-1] > x:
                j = j - 1
            a.insert(j, x)
```



So, as **i** increases, the potential number of comparisons also increases

The label next to a row is the value of **i** passed to the **moveLeft** function

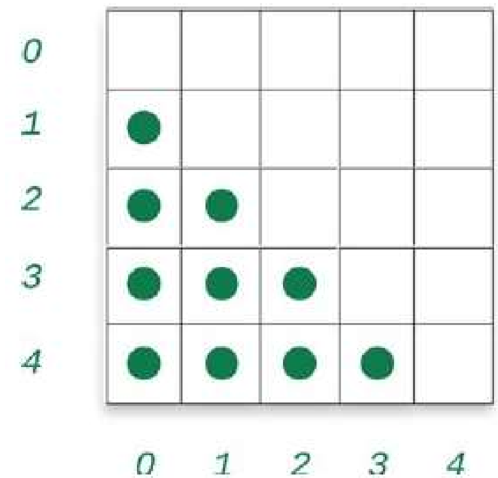
Nested loops

The diagram below is for a call to **isort** with a list of n=5 items

There are 4+3+2+1=10 dots total, signifying that there we be at most 10 comparisons necessary

In general, for a list with n items, the potential number of comparisons is $n(n-2)/2 \approx n^2/2$

We say that in the worst case, the sorting algorithm will make approximately $n^2 / 2$ comparisons



A question about isort

- How many comparisons will be made when **isort** is passed a list that is already sorted, such as this list of 10 items?

  a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

- Consider the while loop's condition:

  while j > 0 and a[j-1] > x:

- The part **a[j-1] > x** will never be true! Do you see why?
  - This means that the while-loop's body will never execute
  - Therefore, the algorithm will compare **a[j-1] > x** exactly 9 times because the outer loop will repeat exactly 9 times for this particular list:

    for i in range(1, len(a)):

    len(a) == 10 here

A question about isort

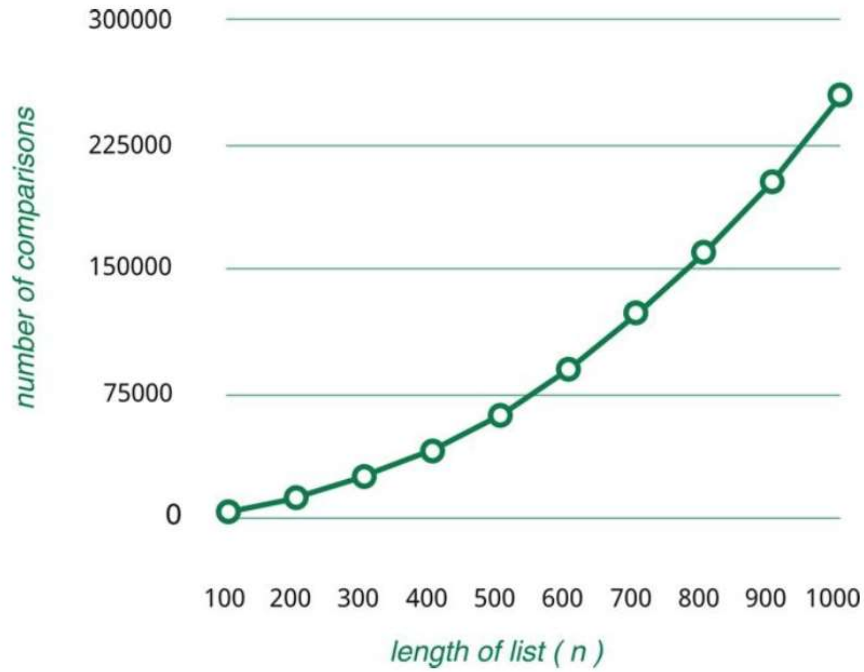
- In general, for a list of n items that is already sorted, how many comparisons will the program make?
- The outer loop will repeat exactly *n*-1 times, and, for each of those iterations, perform the comparison **`a[j-1] > x`** exactly once
- Therefore, the code will perform exactly *n*-1 of those **`a[j-1] > x`** comparisons
- This is the kind of **algorithm analysis** that computer scientists frequently do, so let's look at this topic in a little more detail

Estimating the # of comparisons

- So, the formula for the worst case number of comparisons in isort is: $n(n-1)/2 \approx n^2 – n/2$
- For small lists, we can compute the exact answer: $(5^2-5)/2 = 20/2 = 10$
- For larger lists, the *–n* term doesn't affect the result very much because n^2 will be much larger than *–n*
- We say that the n^2 term *dominates* the expression
- Therefore, we can get a good estimate by computing only $n^2 / 2$

Length of list vs. # of comparisons

- This graph gives a sense of how much “work” the insert sort algorithm does, based on the length of the input list



Big-Oh notation

- Computer scientists use the notation *$O(n^2)$* to mean “for large n, the number of comparisons will be roughly *n^2*”
- *$O(n^2)$* is read aloud as “oh of n-squared”
  - Or sometimes “big oh of n-squared”
  - Or sometimes “order n-squared”
- There is a precise definition of what it means for an algorithm to be *$O(n^2)$*, but for this course we’ll just use the notation informally
- For **`isort`**, the notation means “on the order of *n^2* comparisons”
- Thus, insertion sort is a *$O(n^2)$ algorithm*

Big-Oh notation

- What about linear search? How efficient is that algorithm?
- Like insertion sort, linear search performs many comparisons
- In the worst case, the linear search algorithm won't find the desired item beause the item is not in the list
- In that case, the algorithm will need to inspect every one of the n items in the list
- Therefore, we say that linear search is an *O(n)* algorithm
  - "oh of n"
  - "big oh of n"
  - "ofrder n"
  - All equivalent ways of expressing the efficiency

Scalability

- The fact that the number of comparisons grows as the square of the list size may not seem important
  - For small-to-moderate-sized lists it’s not a big deal
  - But execution time will start to be a factor for larger lists
- The ability of an algorithm to solve increasingly larger problems is an attribute known as **scalability**
  - We say that an efficient algorithm *scales well* for larger inputs
- We’ll revisit this idea after looking at more sophisticated sorting algorithms in a future lecture of the course

Selection sort

- The selection sort algorithm is another *O(n^2)* iteration-based algorithm for sorting a list of values:
  1. Find the smallest value. Swap it (exchange it) with the first value in the list.
  2. Find the second-smallest value. Swap it with the second value in the list.
  3. Find the third-smallest value. Swap it with the third value in the list.
  4. Repeat finding the next-smallest value and swapping it into the correct position until the list is sorted.
- Let's see some examples of how this algorithm works
- Also, be sure to check out visualgo.net/en/sorting

Selection sort: example #1

```
7 1 6 9 5 4
1 7 6 9 5 4     swapped 1 and 7
1 4 6 9 5 7     swapped 4 and 7
1 4 5 9 6 7     swapped 5 and 6
1 4 5 6 9 7     swapped 6 and 9
1 4 5 6 7 9     swapped 7 and 9
```

- Eventually, only the largest value will remain
  - But, it will be in the rightmost position, so we don’t need to do anything with it

Selection sort: example #2

```
8 4 6 7 5 3 2 9 1
1 4 6 7 5 3 2 9 8     swapped 1 and 8
1 2 6 7 5 3 4 9 8     swapped 2 and 4
1 2 3 7 5 6 4 9 8     swapped 3 and 6
1 2 3 4 5 6 7 9 8     swapped 4 and 7
1 2 3 4 5 6 7 9 8     swapped 5 and 5 (?)
1 2 3 4 5 6 7 9 8     swapped 6 and 6 (?)
1 2 3 4 5 6 7 9 8     swapped 7 and 7 (?)
1 2 3 4 5 6 7 8 9     swapped 8 and 9
```

- We note that sometimes the algorithm does no useful work, like “swapping” 5 with itself

Selection sort

- Perhaps you noticed that during execution of the algorithm, the list is divided into two parts:
  - the sorted part (green)
  - the yet-to-be-sorted part (black)
- Also you may have noticed that the largest element winds up in the rightmost spot without any additional work
- Think about that for a moment. Suppose we have 10 elements in our list.
  - Once we have moved the 9 smallest elements into their final positions, the 10th (largest) value *must* be in the rightmost position
  - This has a small implication in the implementation

Selection sort

- Python makes it very easy to swap (exchange) the values stored in two variables
- To exchange the contents of two variables **`x`** and **`y`**, all we need to type is this: **`x, y = y, x`**
- This swapping notation also works with elements of a list
- Suppose **`i`** and **`j`** are valid indices of list **`a`**
- We can type this to swap the contents of **`a[i]`** and **`a[j]`**:

```
a[i], a[j] = a[j], a[i]
```

- With the pseudocode from earlier and this syntax for swapping list elements, we can implement selection sort

Example: selection_sort.py

```
def selection_sort(a):
    for i in range(len(a)-1):
        least_i = i
        for k in range(i+1, len(a)):
            if a[k] < a[least_i]:
                least_i = k
        a[least_i], a[i] = a[i], a[least_i]
```

- See selection_sort.py for fully commented code and additional explanation

Example: Lucky numbers

- Define a *lucky number* as a positive integer whose decimal (base 10) representation contains only the lucky digits 4 and 7
- For example, numbers 47, 744, 4 are lucky, whereas 5, 73 and -3 are not
- Consider a function **is_lucky_number(num)** that returns **True** if **num** is a lucky number and **False**, otherwise
- We need to extract each digit one at a time and inspect it
  - If the digit is neither 4 nor 7, the number is not lucky
  - Otherwise, it is 4 or 7, so discard it and move to the next digit, stopping when we run out of digits

Example: lucky.py

```
def is_lucky_number(num):
    if num <= 0:
        return False
    while num > 0:
        if num % 10 == 4 or num % 10 == 7:
            num //= 10 # discard the digit
        else:
            return False
    return True
```

- See lucky.py for fully commented code and additional explanation

Example: Almost-lucky numbers

- Define an *almost-lucky number* as a positive integer that is divisible by a lucky number
- For example, 611 is almost-lucky because it is divisible by the lucky number 47
- Consider a function **almost_lucky_divisor(num)** that returns the *largest* lucky number that divides evenly into **num** if **num** is an almost-lucky number
    - The function returns **None** if **num** is not an almost-lucky number
- Note that every lucky number is an almost-lucky number because every lucky number is divisible by itself

Example: almost_lucky_divisor.py

```
def almost_lucky_divisor(num):
   for divisor in range(num, 0, -1):
      if num % divisor == 0 and
                     is_lucky_number(divisor):
         return divisor
   return None
```

Questions?