CMPT 120
Algorithms

Summer 2012

Instructor: Hassan Khosravi

Searching

- Searching is an important program in computing

- “Searching” is the problem of looking up a particular value in a list or other data structure. You generally want to find the value (if it’s there) and determine its position.
  - We will only worry about searching in lists here. There are many other data structures that can be used to store data; each one has its own searching algorithms that can be applied

Linear Search

- For lists in general, you have to look through the whole list to determine if the value is present or not.
  - search through the list from element 0 to the end
  - if you find the value you're looking for return it's index value. If you never do, return −1. (We will always use the "position" −1 to indicate "not found".)
  - This search algorithm is called linear search

```
def search(lst, val):
    """
    Find the first occurrence of val in lst. Return its
    index or -1 if not there.
    >>> search([0, 10, 20, 30, 40], 30)
    3
    >>> search([0, 10, 20, 30, 40], 25)
    -1
    """
    for i in range(len(lst)):
        if lst[i]==val:
            # we only care about the first match,
            # so if we've found one, return it.
            return i
    # if we get this far, there is no val in lst.
    return -1
```

- What's the running time of a linear search for a list with n items?
  - At worst, it will have to scan through each of the n elements, checking each one. So, the running time is n.
  - We can do better if the list is arranged properly.

Binary Search

- If you are looking through a phone book, we don't read the whole phonebook.
  - They are sorted by name, so you don't have to scan every entry
- if we have a Python list that's in order, we should be able to take advantage of this to search faster.
- This algorithm has a lot in common with the guessing game
  - Search([1, 2, 3, ..., 100], guess).

Binary search.

```
def binary_search(lst, val):
    """
    Find val in lst. Return its index or -1 if not there.
    The list MUST be sorted for this to work.
    >>> binary_search([2, 4, 5, 6, 24, 100, 1001], 100)
    5
    >>> binary_search([2, 4, 5, 6, 24, 100, 1001], 10)
    -1
    >>> binary_search([2, 4, 5, 6, 24, 100, 1001], 2000)
    -1
    """
```

Binary search

```
# keep track of the first and last possible positions.
    first = 0
    last = len(lst)-1
    while first <= last:
        mid = (first+last)/2
        if lst[mid] == val:
            # found it
            return mid
        elif lst[mid] < val:
            # too small, only look at the right half
            first = mid+1
        else: # lst[mid] > val
            # too large, only look at the left half
            last = mid-1
    # if we get this far, there is no val in lst.
    return -1
```

- To use binary search, you have to keep the list in sorted order. That means that you can't just use list.append() to insert something into the list anymore.
- Inserting into a sorted list takes up to n steps because Python has to shuffle the existing items in the list down to make room.

- keeping a sorted list and doing binary search is only worthwhile if you need to search a lot more than you insert.
  - If you have some data that doesn't change very often, but need to find values regularly, it's more efficient to keep the list sorted because it makes searches so much faster.

- Searching is covered in more detail in CMPT 225 and 307

Binary_append

```
def binary_append(lst,val):
# keep track of the first and last possible positions.
    first = 0
    last = len(lst)-1
    while first <= last:
        mid = (first+last)/2
        print mid
        if lst[mid] == val or first==last:
            if lst[mid] <= val:
              lst[mid:mid+1] = [lst[mid],val]
              print True
            else:
               lst[mid:mid+1] = [val,lst[mid]]
               print False
            return
        elif lst[mid] < val:
            # too small, only look at the right half
            first = mid+1
        else: # lst[mid] > val
            # too large, only look at the left half
            last = mid-1
    # if we get this far, there is no val in lst.
    return -1

p=[2, 4, 5, 6, 24, 100, 1001]
binary_append(p, 101)
print p

#[2, 4, 5, 6, 24, 100, 1001],101
#[2, 4, 5, 6, 24, 100, 1001], 99
```

Sorting

- Sorting is another important problem in computing science.
  - Searching is much faster if the list is sorted first

- A list in Python can be put in order by calling its sort method:
- >>> mylist = [100, -23, 12, 8, 0]
- >>> mylist.sort()
- >>> print mylist
- [-23, 0, 8, 12, 100]

Repeated letters without sorting

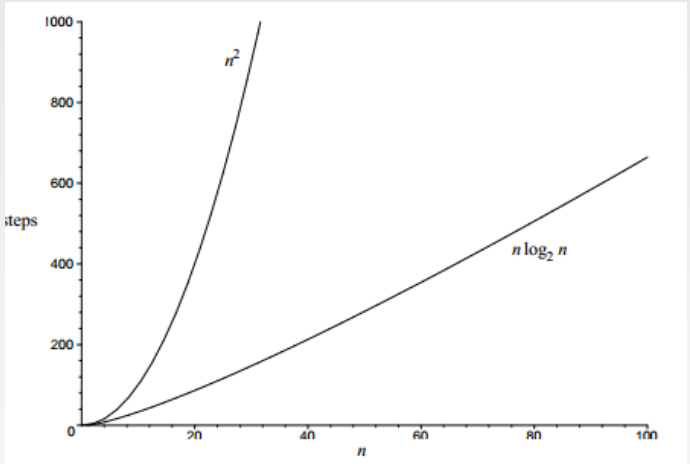
```
word = raw_input("Enter the word: ")
counter = 0
length = len(word)
for i in range(length): # for each letter in the word...
    for j in range(i+1, length):  # for each letter after that one...
        if word[i]==word[j]:
            counter = counter + 1
if counter>0:
    print "There are repeated letters"
else:
    print "There are no repeated letters"
```

- First sorts the letters of the string. Then, any identical letters will be beside each other. So, to check for repeated letters, you only have to skim through the characters once

Example: Repeated letters with sorting

```
word = raw_input("Enter the word: ")
counter = 0
letters = list(word) # converts to a list of characters.
# 'gene' becomes ['g','e','n','e']
letters.sort() # now identical letters are adjacent
# above becomes ['e','e','g','n']
for i in range(len(word)-1):
    if letters[i]==letters[i+1]:
        counter = counter +1
if counter >0:
    print "repeated words"
else:
    print "no repeated words"
```

- The sort method has running time n log n on a list with n elements. The rest of the program just scans once through the list, so it takes n steps. The total running time will be n log n + n.
  - Removing the lower-order terms, we get n log n



How to sort

- Sorting is important enough that you should have some idea of how it's done.
- The sort method of a list has running time n log n.
- Algorithms that sort in n log n steps are fairly complicated and will have to wait for another course

- Suppose you're given a pile of a dozen exam papers and are asked to put them in order by the students'
  - Flip through the pile and find the paper that should go first.
  - Put that paper face-down on the table.
  - Repeat from step 1 with the remaining pile, until there are no papers left.

  - The idea behind selection sort

Iteration	Initial List	Operation
0	6, 2, 8, 4, 5, 3	Swap 2 and 6
1	**2,** 6, 8, 4, 5, 3	Swap 6 and 3
2	**2, 3,** 8, 4, 5, 6	Swap 8 and 4
3	**2, 3, 4,** 8, 5, 6	Swap 8 and 5
4	**2, 3, 4, 5**, 8, 6	Swap 8 and 6
5	**2, 3, 4, 5**, **8**, **6**	Do nothing

- This algorithm has running time n^2.

```
for every element e from the list,
    for every element f from e to the end of the list,
        if f < smallest,
            set smallest to f
    swap smallest and e
```

```
def selection_sort(lst):
    """
    Sort lst in-place using selection sort
    """
    for pos in range(len(lst)):
        # get the next smallest in lst[pos]
        # find the next smallest
        small = lst[pos] # smallest value seen so far
        smallpos = pos # position of small in lst
        for i in range(pos+1, len(lst)):
            # check each value, searching for one
            # that's smaller than the current smallest.
            if lst[i] < small:
                small = lst[i]
                smallpos = i
         # swap it into lst[pos]
        lst[pos], lst[smallpos] = lst[smallpos], lst[pos]
```

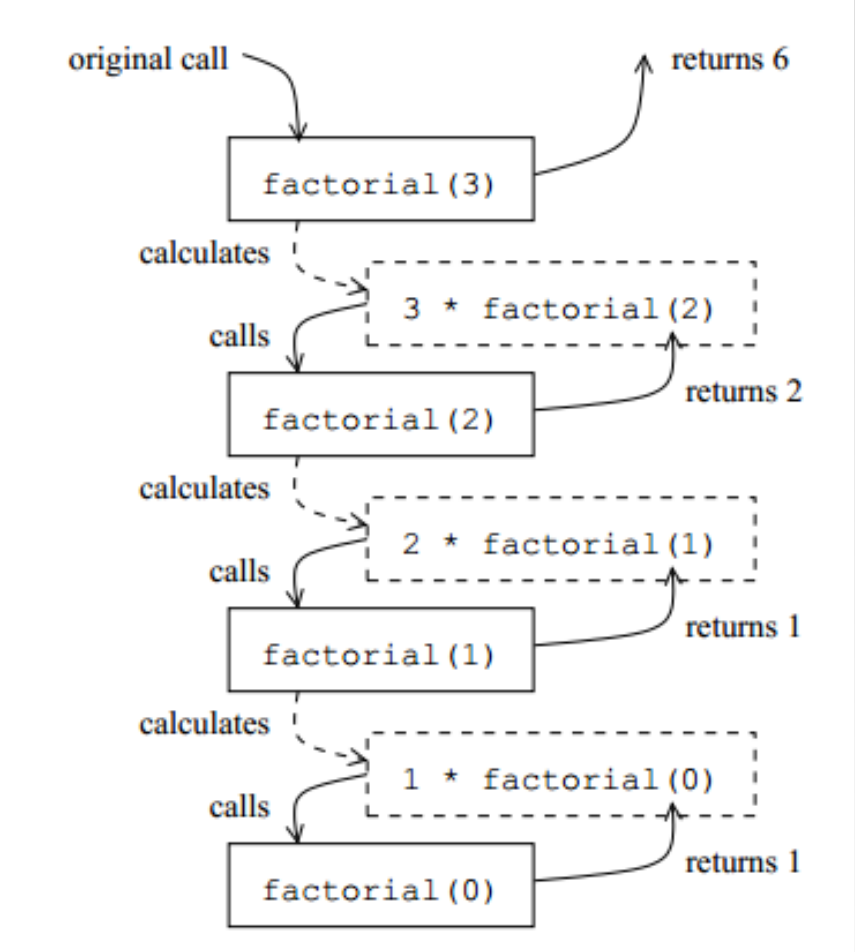
Recursion

- It's possible for a function to call another function.
- It's also possible for a function to call itself
  - This technique is called recursion
  - Many algorithms are most easily described recursively

- Consider calculating the factorial of a number

$$n! = \begin{cases} 1 & \text{for } n = 0 \\ n \times (n-1)! & \text{for } n > 0 \end{cases}$$

- We can use this same definition to create a Python function that calculates the factorial of a (positive) integer.
- def factorial(n):
- """
- Calculate n! recursively.
- >>> factorial(10)
- 3628800
- >>> factorial(0)
- 1
- """
- if n==0:
- return 1
- else:
- return n * factorial(n-1)


original call
returns 6
factorial(3)
calculates
3 * factorial(2)
calls
factorial(2)
returns 2
calculates
2 * factorial(1)
calls
factorial(1)
returns 1
calculates
1 * factorial(0)
calls
factorial(0)
returns 1

How it works

- Whenever a function calls itself, you should think of a new copy of the function being made.
  - if we call factorial(3), while running, it will call factorial(2). This will be a separate function call, with separate arguments and local variables.
  - It will run totally independently of the other instance of the function.
  - factorial(3) calls factorial(2), which itself calls factorial(1), which calls factorial(0).

Understanding Recursion

- When looking at a recursive algorithm, many people find it too complicated to think of every function call
  - Keeping track of every step of the recursion isn't really necessary to believe that the function works
    - Factorial(100)
  - Keeping track of a few steps is usually enough to understand the algorithm
  - Make sure that the function and its recursive calls are structured so that recursive calls head towards the base case that ends the recursion.

Designing with Recursion

- Step 1: Find a Smaller Subproblem
  - The whole point of making a recursive call is to solve a similar, but smaller problem.
    - n! = n × (n − 1)!
    - if we can somehow calculate (n − 1)! it's easy to use that to calculate n!
  - To reverse the string, we have to ask: if we reverse part of the string, can we use that to finish reversing the whole string?
    - If we reverse the tail of the string (string[1:])
    - if we are trying to reverse the string "looter", the tail (string[1:]) is or "ooter"
    - . If we make a recursive call to reverse this (reverse(string[1:])), it should return "retoo".

Designing with Recursion

- Step 2: Use the Solution to the Sub problem
  - Once you found a subproblem that you can work with, you can get the result with a recursive call to the function
  - In the factorial example, we know that once we have calculated $(n-1)!$, we can simply multiply by n to get $n!$.
    - n * factorial(n-1).
  - For reversing a string, again using "looter" as an example, reverse(string[1:]) returns "retoo" and string[0] is "l". The whole string reversed is:
    - reverse(string[1:]) + string[0]

Designing with Recursion

- Step 3: Find a Base Case
  - There will be a few cases where a recursive call would not work

$$n! = \begin{cases} 1 & \text{for } n = 0 \\ n \times (n-1)! & \text{for } n > 0 \end{cases}$$

  - Typically these will be the smallest cases where it's not possible to subdivide the problem further.
  - These cases can simply be handled with an if statement.
    - If the arguments point to a base case, return the appropriate result. Otherwise, proceed with the recursive solution as designed above.
    - In the factorial example, the identity n! = n × (n − 1)! isn't true for n = 0

Designing with Recursion

- For reversing a string, there is also one case where the method outlined above for the recursive case can’t be followed.
  - What should be the result when we call reverse("")?
    - reverse(string[1:]) + string[0] would raise an error

- We should also check other small cases:
  - what is the reverse of a singlecharacter string?
  - The function call reverse("X") should return "X“
    - reverse(string[1:]) + string[0] == reverse("") + "X“

Designing with Recursion

- Every recursive call MUST eventually get to a base case
- If this isn't the case, the function will keep making more and more recursive calls without ever stopping.
  - Python will stop when the recursion passes a certain "depth". It will give the error "maximum recursion depth exceeded".

- Step 4: Combine the Base and Recursive Cases
  - Once we have identified the base case(s) and what recursive calculation to do in other cases, we can write the recursive function.
    - check to see if the argument(s) point to a base case
      - If so, just return the solution for this case.
    - If we don't have a base case, then the recursive case applies

if we have a base case,

    return the base case solution

otherwise,

    set *rec_result* to the result of a recursive call on the subproblem
    return the solution, built from *rec_result*

Example reversing a string

- Find a Smaller Subproblem
  - If you can reverse everything except the first character, it is easy to reverse the whole string
    - String "looter“ → (string[1:]) is or "ooter“ → retoo → retool
- Step 2: Use the Solution to the Sub problem
  - reverse(string[1:]) + string[0]
- Step 3: Find a Base Case
  - For “” return “”

```
def reverse(string):
    """
    Return the reverse of a string
    >>> reverse("bad gib")
    'big dab'
    >>> reverse("")
    ''
    """
    if len(string)==0:
        # base case
        return ""
    else:
        # recursive case
        rev_tail = reverse(string[1:])
        return rev_tail + string[0]


print reverse("asdf")
```

Debugging Recursion

- The first thing that should be done when testing or trying to find errors is to examine the base case(s).
  - >>> factorial(0)
  - 1
  - >>> reverse("")
  - ''
- Once we know the base cases are working, we can then easily test the cases that call the base case.
  - >>> factorial(1)
  - 1
  - >>> reverse("X")
  - 'X'
  - >>> reverse("(")
  - '('
- Then the cases two steps from the base.

Another Example

- create a function that inserts spaces between characters in a string and returns the result.
  - >>> spacedout("Hello!")
  - ’H e l l o !’
  - >>> spacedout("g00DbyE")
  - ’g 0 0 D b y E’

- [Find a Smaller Subproblem.]
  - "marsh“ → “m a r s” + “ h”
- [Use the Solution to the Subproblem.]
  - Result(input(:-1) + “ “ + input(-1)

- [Find a Base Case.]
  - Empty string
  - String of length 1

```
def spacedout(string):
    """
    Return a copy of the string with a space between
    each character.
    >>> spacedout("abcd")
    'a b c d'
    >>> spacedout("")
    >>> spacedout("Q")
    'Q'
    """
    if len(string) <= 1:
        return string
    else:
        head_space = spacedout(string[:-1])
        return head_space + " " + string[-1]


print spacedout("hassan")
print spacedout("h")
```