Divide and Conquer

1. Guessing a Secret
   - playing divide and conquer
2. Binary Search
   - searching in a sorted list
   - membership in a sorted list
3. Bisection Search
   - inverting a sampled function
   - bisection search in arrays
   - approximating roots by bisection
4. Exercises

MCS 275 Lecture 15
Programming Tools and File Management
Jan Verschelde, 13 February 2017

Divide and Conquer

1. Guessing a Secret
   - playing divide and conquer
2. Binary Search
   - searching in a sorted list
   - membership in a sorted list
3. Bisection Search
   - inverting a sampled function
   - bisection search in arrays
   - approximating roots by bisection
4. Exercises

Guessing a Secret

A little game:

- The computer picks a random integer in [0, 1000].
- It is up to the user to guess the number.

Suppose making a guess costs $1
and you get $100 for the right guess.

Would you play it?

Suppose now the computer would tell you
after each incorrect guess: `too low` or `too high`.

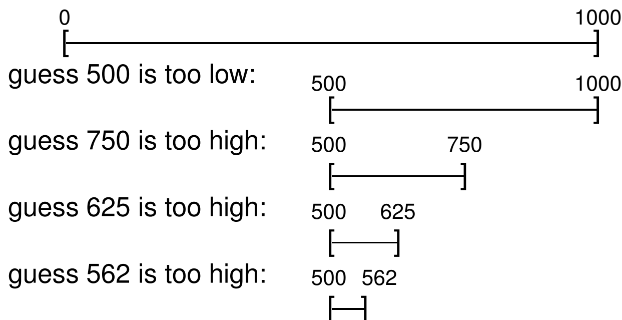
With same cost and price, would you now play it?

playing the guessing game

```
$ python findsecret.py
Guess number in [0, 1000] : 500
Your guess is too low.
Guess number in [0, 1000] : 750
Your guess is too high.
Guess number in [0, 1000] : 625
Your guess is too high.
Guess number in [0, 1000] : 562
Your guess is too high.
Guess number in [0, 1000] : 531
Your guess is too high.
Guess number in [0, 1000] : 516
Your guess is too high.
Guess number in [0, 1000] : 508
Your guess is too high.
Guess number in [0, 1000] : 504
Your guess is too low.
Guess number in [0, 1000] : 506
found 506 after 9 guesses
```

halving the search space

The search space is [0, 1000] at the start.



In each step the search space is cut in half.
After 10 steps, we are down to the last digit.

In every step we recover one bit of the secret.

Divide and Conquer

1. Guessing a Secret
   - playing divide and conquer
2. Binary Search
   - searching in a sorted list
   - membership in a sorted list
3. Bisection Search
   - inverting a sampled function
   - bisection search in arrays
   - approximating roots by bisection
4. Exercises

searching lists with builtin methods `in` and `index`

Let us generate a list of 10 random two digit numbers.

```
>>> from random import randint
>>> L = [randint(10, 99) for _ in range(10)]
>>> L
[32, 61, 50, 81, 30, 14, 53, 92, 22, 23]
>>> 10 in L
False
>>> 81 in L
True
>>> L.index(81)
3
>>> L[3]
81
```

`L.index(n)` will throw a `ValueError` if `not n in L`.

linear search

The search problem:

- Input : `L` a list of numbers and some number `x`.
- Output : index `k == -1`, if `not x in L`, otherwise `L[k] == x`.

Algorithm for a linear search:

1. Enumerate in `L` all its elements in `L[k]`.
2. If `L[k] == x` then `return k`.
3. Return `-1` at the end of the loop.

Cost analysis:

- Best case: `x` at the start of `L`.
- Worst case: `x` at the end of `L`.
- On average: $c \times n$ steps, n = `len(L)`, for some constant $c \approx 0.5$. We say its cost is $O(n)$.

searching *linearly* in sorted list

To sort a list `L`, do `L.sort()`.

```
def linear_search(numbers, nbr):
    """
    Returns -1 if nbr belongs to numbers,
    else returns k for which numbers[k] == nbr.
    Items in the list numbers must be sorted
    in increasing order.
    """
    for i in range(len(numbers)):
        if numbers[i] == nbr:
            return i
        elif numbers[i] > nbr:
            return -1
    return -1
```

Divide and Conquer

1. Guessing a Secret
   - playing divide and conquer
2. Binary Search
   - searching in a sorted list
   - membership in a sorted list
3. Bisection Search
   - inverting a sampled function
   - bisection search in arrays
   - approximating roots by bisection
4. Exercises

membership test for a sorted list

The builtin `in` and `index` for lists do not exploit order.
To sort a list `L`, do `L.sort()`.

Searching in a sorted list:

Input : `L` is a list of numbers, ordered increasingly;
`x` is some number.

Output : `True` if `x in L`, `False` otherwise.

Applying divide and conquer:

- Base cases: `len(L) == 0` or `len(L) == 1`.
- Let `m = len(L)//2`, if `L[m] == x`, `return True`.
- If `x < L[m]`: search in first half of `L`: `L[:m]`.
- If `x > L[m]`: search in second half of `L`: `L[m+1:]`.

code for a binary search

```
def binary_search(numbers, nbr):
    """
    Returns True if nbr is in the sorted numbers.
    Otherwise False is returned.
    """
    if len(numbers) == 0:
        return False
    elif len(numbers) == 1:
        return numbers[0] == nbr
    else:
        middle = len(numbers)//2
        if numbers[middle] == nbr:
            return True
        elif numbers[middle] > nbr:
            return binary_search(numbers[:middle], nbr)
        else:
            return binary_search(numbers[middle+1:], nbr)
```

tracing the binary search

We trace the search by

- accumulating the depth of the recursion,
- printing as many spaces as the depth,
- printing the remaining list to search in.

```
Give lower bound : 10
Give upper bound : 99
How many numbers ? 10
L = [10, 14, 14, 19, 20, 38, 53, 60, 66, 72]
Give number to search for : 21
find 21 in L = [10, 14, 14, 19, 20, 38, 53, 60, 66, 72]
 find 21 in L = [10, 14, 14, 19, 20]
  find 21 in L = [19, 20]
   find 21 in L = []
21 does not occur in L
```

a binary search for the index

The builtin `index` does not exploit order either.

```
def binary_index(numbers, nbr):
    """
    Applies binary search to find the
    position k of nbr in the sorted numbers.
    Returns -1 if not nbr in numbers, or else
    returns k for which numbers[k] == nbr.
    """
```

Same divide and conquer as `binary_search()`,
extra additional care for index calculation.

code for `binary_index`

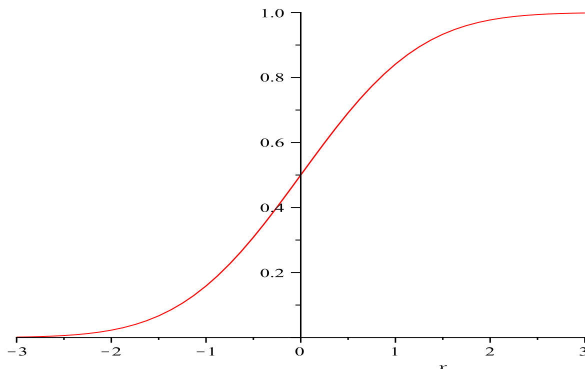
```
if len(numbers) == 0:
    return -1
elif len(numbers) == 1:
    if numbers[0] == nbr:
        return 0
    else:
        return -1
else:
    middle = len(numbers)//2
    if numbers[middle] == nbr:
        return middle
    elif numbers[middle] > nbr:
        return binary_index(numbers[:middle], nbr)
    else:
        k = binary_index(numbers[middle+1:], nbr)
        if k == -1:
            return -1
        return k + middle + 1
```

Divide and Conquer

1. Guessing a Secret
   - playing divide and conquer
2. Binary Search
   - searching in a sorted list
   - membership in a sorted list
3. Bisection Search
   - inverting a sampled function
   - bisection search in arrays
   - approximating roots by bisection
4. Exercises

inverting a function

Consider a cumulative distribution function, for example:



Given 1. a sampled array A of function values,

2. a particular function value $y = f(x)$.

Find k: $A[k] \leq y \leq A[k+1]$.

sorted arrays

```
>>> from random import uniform as u
>>> L = [u(-1, 1) for _ in range(10)]
>>> L.sort()
>>> from array import array
>>> A = array('d', L)

def linear_search(arr, nbr):
    """
    Returns the index k in the array arr
    such that arr[k] <= nbr <= arr[k+1].
    A must be sorted in increasing order.
    """
    for i in range(len(arr)):
        if nbr <= arr[i]:
            return i-1
    return len(arr)
```

Divide and Conquer

1. Guessing a Secret
   - playing divide and conquer
2. Binary Search
   - searching in a sorted list
   - membership in a sorted list
3. Bisection Search
   - inverting a sampled function
   - bisection search in arrays
   - approximating roots by bisection
4. Exercises

bisection search in arrays

```
def bisect_search(arr, nbr):
    """
    Returns the index k in the array arr such that
    arr[k] <= nbr <= arr[k+1] applying binary search.
    """
```

Base cases:

1. `len(arr) == 0`: `return -1`
2. `len(arr) == 1`: if `arr[0] <= nbr`: `return 0`,
   otherwise `return -1`

General case: `m = len(arr)//2`

1. if `nbr < arr[m]`: search in `arr[:m]`
2. if `nbr > arr[m]`: search in `arr[m+1:]`
   Add `m+1` to the index returned by 2nd search.

code for bisect_search

```
def bisect_search(arr, nbr):

    if len(arr) == 0:
        return -1
    elif len(arr) == 1:
        if arr[0] <= nbr:
            return 0
        else:
            return -1
    else:
        middle = len(arr)//2
        if nbr < arr[middle]:
            return bisect_search(arr[:middle], nbr)
        else:
            k = bisect_search(arr[middle+1:], nbr)
            return k + middle +1
```

Divide and Conquer

1 Guessing a Secret
- playing divide and conquer

2 Binary Search
- searching in a sorted list
- membership in a sorted list

3 Bisection Search
- inverting a sampled function
- bisection search in arrays
- approximating roots by bisection

4 Exercises

the bisection method

Let f be a continuous function over $[a, b]$, and $f(a)f(b) < 0$,
then $f(r) = 0$, for some $r \in [a, b]$.

Key steps in the bisection method:

1. Let $m = \frac{a+b}{2}$.
2. If $f(a)f(m) < 0$,
   then replace $[a, b]$ by $[a, m]$,
   otherwise replace $[a, b]$ by $[m, b]$.

Every step gains one bit in an approximate root r of f.

the function `bisect`

```
def bisect(fun, left, right):
    """
    If (left, right) contains a root of fun,
    then on return is a smaller (left, right)
    containing a root of fun.
    """
    midpoint = (left + right)/2
    if fun(left)*fun(midpoint) < 0:
        return (left, midpoint)
    else:
        return (midpoint, right)
```

The accuracy of the root is `right - left`.
If `right - left < tol`, return `(left, right)` else `bisect`.

applying bisection recursively

The accuracy of the root is `right - left`.
If `right - left < tol`, return `(left, right)` else `bisect`.

```
def bisectroot(fun, left, right, tol):
    """
    Continues bisecting till the right - left
    is less than tol.
    """
    if right-left < tol:
        return (left, right)
    else:
        (left, right) = bisect(fun, left, right)
        return bisectroot(fun, left, right, tol)
```

approximating $\sqrt{2}$

```
$ python bisection.py
Give a function in x : x**2 - 2
give left bound A : 1
give right bound B : 2
give the tolerance : 1.0e-12
A =  1.41421356237242 43
B =  1.41421356237333 38
$
```

Exercises

1. Write an iterative version of `binary_search`.
2. Write an iterative version of `bisectroot`.
3. The minimum of a list of unsorted numbers is the minimum of the minimum of the first half and the minimum of the second half of the list. Write a function to compute the minimum this way.
4. Given is a list of lexicographically sorted names.
   Use divide and conquer to find the name that occurs most frequently in the list.
5. Develop bisection search to compute the binary representation of a number `x`, starting at the most significant bit.
   Use `math.log(x,2)` to compute the total number of bits needed to represent `x`.