Algorithms Design & Recursion

CS177 – Recitation 15
Agenda

• What’s an Algorithm.
• Search algorithms
  • Linear search
  • Binary search
• Recursion.
An algorithm is a step-by-step list of instructions to solve a problem.

- An algorithm is like a recipe.

### Best Brownies

**Directions**

1. Preheat oven to 350 degrees F (175 degrees C). Grease and flour an 8-inch square pan.
2. In a large saucepan, melt 1/2 cup butter. Remove from heat, and stir in sugar, eggs, and 1 teaspoon vanilla. Beat in 1/3 cup cocoa, 1/2 cup flour, salt, and baking powder. Spread batter into prepared pan.
3. Bake in preheated oven for 25 to 30 minutes. Do not overcook.
4. To Make Frosting: Combine 3 tablespoons softened butter, 3 tablespoons cocoa, honey, 1 teaspoon vanilla extract, and 1 cup confectioners’ sugar. Stir until smooth. Frost brownies while they are still warm.
Another example for an algorithm: Tying the Windsor Knot
Given a List of Numbers, how would you find a particular number from that list?

Find 5
Given a List of Numbers, how would you find a particular number from that list

Find 5
Given a List of Numbers, how would you find a particular number from that list

Find 5
Given a List of Numbers, how would you find a particular number from that list?

Find 5

Return 2
Keep going through the elements one by one till you find your match. This process is called “Sequential Search” or “Linear Search”
def seqSsearch(nums, n):
    for i in range(len(nums)):
        if nums[i] == n:
            return i
    return -1
Is Sequential Search the best way?

• What happens if you are searching among very big number of elements?

• There are also many algorithms solving the same problem.
• We want a good algorithm. But what defines “goodness”? 
We use Space Complexity and Time Complexity when evaluating an algorithm

- **Space complexity**: How much memory the algorithm needs? In other words, how many variables the algorithm needs?
- **Time complexity**: The number of steps executed by the algorithms?
- Why not just measure the time the algorithm takes!?
  - Different machines, architectures \(\rightarrow\) different execution times!

We need to express the space/time complexity in terms of the data size. For example: the size of the list we search in.
Space Complexity for Sequential Search is constant

```python
def seqSsearch(nums, n):
    for i in range(len(nums)):
        if nums[i] == n:
            return i
    return -1
```

- If len(nums) equals 5, this algorithm will use only one variable (i).
- If len(nums) equals 5000, this algorithm will STILL use only one variable (i).
- This means the number of variables this algorithm uses is constant with respect the number of elements we process.
- The space complexity of this algorithm is constant.
Time Complexity for Sequential Search is linear

```python
def seqSsearch(nums, n):
    for i in range(len(nums)):
        if nums[i] == n:
            return i
    return -1
```

- If `len(nums)` equals 5, this algorithm will check the if condition 5 times.
- If `len(nums)` equals 5000, this algorithm will check the if condition 5000 times.
- This means the number of times the if condition is evaluated depends on the number of elements we process.
- The space complexity of this algorithm is linear with the size of the data.

Checking if two numbers are equal or not is the core operation of this algorithm.
If the given list of numbers is sorted, then we can use Binary Search algorithm
def bsearch(x, nums):
    low = 0
    high = len(nums) - 1
    while low <= high:
        mid = (low+high)//2
        item = nums[mid]
        if x == item:
            return mid
        elif x < item:
            high = mid - 1
        else:
            low = mid + 1
    return -1

Find 5
```python
def bsearch(x, nums):
    low = 0
    high = len(nums) - 1
    while low <= high:
        mid = (low + high) // 2
        item = nums[mid]
        if x == item:
            return mid
        elif x < item:
            high = mid - 1
        else:
            low = mid + 1
    return -1
```

**Working of Binary search**

1. low = 0
2. high = 0
3. low <= high
4. mid = (0 + 7) // 2 = 3
5. item = nums[3] = 5
6. x = 5
7. if x == item: return mid (return 3)

```
[0 1 2 3 4 5 6 7 8]
```

Find 5: low=0, high=7

```
[0 1 2 3 4 5 6 7 8]
```
def bsearch(x, nums):
    low = 0
    high = len(nums) - 1
    while low <= high:
        mid = (low + high) // 2
        item = nums[mid]
        if x == item:
            return mid
        elif x < item:
            high = mid - 1
        else:
            low = mid + 1
    return -1
def bsearch(x, nums):
    low = 0
    high = len(nums) - 1
    while low <= high:
        mid = (low+high)//2
        item = nums[mid]
        if x == item:
            return mid
        elif x < item:
            high = mid - 1
        else:
            low = mid + 1
    return -1
def bsearch(x, nums):
    low = 0
    high = len(nums) - 1
    while low <= high:
        mid = (low + high) // 2
        item = nums[mid]
        if x == item:
            return mid
        elif x < item:
            high = mid - 1
        else:
            low = mid + 1
    return -1
def bsearch(x, nums):
    low = 0
    high = len(nums) - 1
    while low <= high:
        mid = (low + high) // 2
        item = nums[mid]
        if x == item:
            return mid
        elif x < item:
            high = mid - 1
        else:
            low = mid + 1
    return -1
Working of Binary search

```python
def bsearch(x, nums):
    low = 0
    high = len(nums) - 1
    while low <= high:
        mid = (low + high) // 2
        item = nums[mid]
        if x == item:
            return mid
        elif x < item:
            high = mid - 1
        else:
            low = mid + 1
    return -1
```

Find 5

mid=5
low=4 high=7

0 1 2 3 4 5 6 7 8
def bsearch(x, nums):
    low = 0
    high = len(nums) - 1
    while low <= high:
        mid = (low + high) // 2
        item = nums[mid]
        if x == item:
            return mid
        elif x < item:
            high = mid - 1
        else:
            low = mid + 1
    return -1

# Working of Binary search

Find 5

mid=5
low=4  high=7

item = nums[mid] = 6
def bsearch(x, nums):
    low = 0
    high = len(nums) - 1
    while low <= high:
        mid = (low + high) // 2
        item = nums[mid]
        if x == item:
            return mid
        elif x < item:
            high = mid - 1
        else:
            low = mid + 1
    return -1
def bsearch(x, nums):
    low = 0
    high = len(nums) - 1
    while low <= high:
        mid = (low + high) // 2
        item = nums[mid]
        if x == item:
            return mid
        elif x < item:
            high = mid - 1
        else:
            low = mid + 1
    return -1
def bsearch(x, nums):
    low = 0
    high = len(nums) - 1
    while low <= high:
        mid = (low + high) // 2
        item = nums[mid]
        if x == item:
            return mid
        elif x < item:
            high = mid - 1
        else:
            low = mid + 1
    return -1
def bsearch(x, nums):
    low = 0
    high = len(nums) - 1
    while low <= high:
        mid = (low + high) // 2
        item = nums[mid]
        if x == item:
            return mid
        elif x < item:
            high = mid - 1
        else:
            low = mid + 1
    return -1

Working of Binary search

Find 5

mid=4
low=4
high=5

item = nums[mid] = 5
def bsearch(x, nums):
    low = 0
    high = len(nums) - 1
    while low <= high:
        mid = (low+high)//2
        item = nums[mid]
        if x == item:
            return mid
        elif x < item:
            high = mid - 1
        else:
            low = mid + 1
    return -1
The time complexity for Binary Search is logarithmic and space complexity is constant

- In each iteration, search space is reduced by half.
  - Initially, search in 8 numbers (1~8)
  - Then, search in 4 numbers (5~8)
  - Finally, search in one number (5)
  - The number of iterations is \( \log_2(\text{len(nums)}) = 3 \)
  - **Logarithmic time complexity**

- Use four variables: low, high, mid, item
  - Independent of \( \text{len(nums)} \)
  - **Constant space complexity**

```python
def bsearch(x, nums):
    low = 0
    high = len(nums) - 1
    while low <= high:
        mid = (low+high)//2
        item = nums[mid]
        if x == item:
            return mid
        elif x < item:
            high = mid - 1
        else:
            low = mid + 1
    return -1
```
Ok.... So what?

- Have you heard about the buzzword "BigData"?
- What if you are asked to search in a list of a billion numbers?

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear search</td>
<td>Run billions of steps</td>
</tr>
<tr>
<td>Binary search</td>
<td>several dozen steps</td>
</tr>
</tbody>
</table>

Win!
Recursion is the process of repeating items in a self-similar way.
Recursion Example: Printing all numbers from n to 0

```python
def print_recursive(n):
    if n < 0:
        return
    print(n)
    print_recursive(n-1)

print_recursive(5)
```

Output:
5
4
3
2
1
0
Calculating Factorial

- Given that Factorial (1) = Factorial (0) = 1
- Factorial (5) = 5 * 4 * 3 * 2 * 1 = 120
- We can write factorial (5) in term of the factorial of smaller numbers:
  - Factorial (5) = 5 * Factorial (4)
    = 5 * 4 * Factorial (3)
    = 5 * 4 * 3 * Factorial (2)
    = 5 * 4 * 3 * 2 * Factorial (1)
    = 5 * 4 * 3 * 2 * 1 = 120

- Generally: Factorial (x) = x * Factorial (x-1)
Calculating Factorial

```
def factorial(x):
    if x < 2:
        return 1
    return x * factorial(x-1)

def main():
    print(factorial(5))
```

Result = 5 * factorial (4)
        4 * factorial (3)
        3 * factorial (2)
        2 * factorial (1)
            * 1

120

The output of the above program is 120
Recursion Analogy with the movie Inception

• You have the function “Dream” 😊

• Each time the function dream calls itself (recursive call), you get into a deeper dream level.

• To wake up from the first dream, you need to wake up from all dreams!

• To wake up you need a kick! The kick in recursion is the return statement.