ALGORITHM DESIGN & ALGORITHM EFFICIENCY

LECTURE 10–1

JIM FIX, REED COLLEGE CSCI 121

COURSE INFO

- **TODAY:** quiz on recursion
- **Today:**
  - ➡ consider algorithm design and algorithm efficiency
  - ➡ case studies:
    - ✦ computing a power
    - ✦ searching a list
- **Next time:**
  - ➡ formalize algorithm efficiency and its notation

WRITING BETTER CODE

- As you become a more sophisticated programmer, you'll be driven to write good code. Some measures of "goodness":
  - Is it ***correct***?
  - Is it ***readable***?
  - Is it ***maintainable***?
  - And sometimes writing ***efficient*** code is important, too.

EFFICIENT CODE

- Efficient code is code that uses fewer resources when run. Examples:
  - ➡ It makes fewer calculations and/or takes fewer steps.
  - ➡ It uses less memory with its data structures.
- For both of these, a program will typically compute its answer faster.
  - It runs faster.

- Today we'll focus on ***running time***.

COMPUTING A POWER OF TWO

```
def power_of2(n):
    i = 0
    x = 1
    while i < n:
        x *= 2
        i += 1
    return x
```

COMPUTING A POWER OF TWO

```
def power_of2(n):
    i = 0
    x = 1
    while i < n:
        x *= 2
        i += 1
    return x
```

- This performs n multiplications to compute 2^n.

COMPUTING A POWER OF TWO

```
def power_of2(n):
    if i == 0:
        return 1
    else:
        return 2 * power_of2(n - 1)
```

COMPUTING A POWER OF TWO

```
def power_of2(n):
    if i == 0:
        return 1
    else:
        return 2 * power_of2(n - 1)
```

- This also performs n multiplications to compute 2^n.

COMPUTING A POWER OF TWO

```
def power_of2(n):
    if i == 0:
        return 1
    elif i % 2 == 0:
        p = power_of2(n // 2)
        return p * p
    else:
        return 2 * power_of2(n - 1)
```

COMPUTING A POWER OF TWO

```
def power_of2(n):
    if i == 0:
        return 1
    elif i % 2 == 0:
        p = power_of2(n // 2)
        return p * p
    else:
        return 2 * power_of2(n - 1)
```

- To compute 2^n this performs between $\log_2(n)$ and $2\log_2(n)$ multiplications.

SEARCHING A LIST

```
def search(value, someList):
    i, n = 0, len(someList)
    while i < n:
        if someList[i] == value:
            return True
        i += 1
    return False
```

SEARCHING A SORTED LIST

Can we do better if a list is sorted?

- Suppose that
  `someList[0]` ≤ `someList[1]` ≤ … ≤ `someList[n-1]`

SEARCHING A SORTED LIST

```
def bsHelp(value, someList, left, right):
    if (right - left + 1) > 0:
        middle = (left + right) // 2
        if value == someList[middle]:
            return True
        elif value < someList[middle]:
            return bsHelp(value,someList,left,middle-1)
        else:
            return bsHelp(value,someList,middle+1,right)
    return False

def binarySearch(value, someList, left, right):
    return bsHelp(value, someList, 0, len(someList))
```

SEARCHING A SORTED LIST

```
def bsHelp(value, someList, left, right):
    if (right - left + 1) > 0:
        middle = (left + right) // 2
        if value == someList[middle]:
            return True
        elif value < someList[middle]:
            return bsHelp(value,someList,left,middle-1)
        else:
            return bsHelp(value,someList,middle+1,right)
    return False
```

- With each `someList[middle]` check…
  - ➡ …we eliminate about half the list items from consideration.
- This means we inspect the list about $\log_2(n)$ times.

SORTING?

- But how much time does it take to sort a list?
  - For next time…