Recursion

Intro2CS – week 6
Designing Algorithms

• There is no single recipe for inventing algorithms

• There are basic rules:
  – Understand your problem well – may require much mathematical analysis!
  – Use existing algorithms (reduction) or algorithmic ideas

• There is a single basic algorithmic technique: Divide and Conquer
Divide and Conquer

• In its simplest (and most useful) form it is simple induction:
  – In order to solve a problem, solve a similar problem of smaller size

• The key conceptual idea:
  – Think only about how to use the smaller solution to get the larger one
  – Do not worry about how to solve to smaller problem (it will be solved using an even smaller one)
Recursion

• A recursive method is a method that contains a call to itself

• Technically:
  – All modern computing languages allow writing methods that call themselves
  – We will discuss how this is implemented later

• Conceptually:
  – This allows programming in a style that reflects divide-and-conquer algorithmic thinking
  – At the beginning recursive programs are confusing – after a while they become clearer than non-recursive variants
A wider perspective

Recursion can be used in:

- Algorithms / Programs ➡ Now
- Definitions ➡ Later today
- Data Structures ➡ Later in the course
- Proofs ➡ Called Induction
def sum_seq(n):
    """ return the sum 1/1 + 1/2 + ... + 1/n """
    if n==0:
        return 0
    else:
        return sum_seq(n-1) + 1/n
Calls and Returns

def sum_seq(n):
    print("called with value", n)
    if n==0:
        print("returning 0")
        return 0
    else:
        answer = sum_seq(n-1) + 1/n
        print("returning", answer)
        return answer

>>> sum_seq(5)
called with value 5
called with value 4
called with value 3
called with value 2
called with value 1
called with value 0
returning 0
returning 1.0
returning 1.5
returning 1.8333333333333333
returning 2.0833333333333333
returning 2.2833333333333333
2.2833333333333333

>>>
Elements of a recursive program

• Basis: a simple case that can be answered without using further recursive calls
  – In our case:  \texttt{if n==0 return 0;}

• Creating the smaller problem, and invoking a recursive call on it
  – In our case:  \texttt{sum_seq(n-1)}

• Using the result on the smaller problem to solve the original problem
  – In our case:  \texttt{return (...) + 1/n}
Theorem: For every integer \( n \geq 0 \), \( \text{sum}_\text{seq}(n) \) returns the value \( \sum_{i=1}^{n}(1/i) \).

Proof: By induction on \( n \):

- **Basis:** for \( n=0 \), \( \text{sum}_\text{seq}(0) \) returns 0 which is correct for an empty sum.

- **Induction step:** When called on \( n \geq 1 \), \( \text{sum}_\text{seq} \) calls \( \text{sum}_\text{seq}(n-1) \), which by the induction hypothesis returns \( \sum_{i=1}^{n-1}(1/i) \). The returned value is thus \( \sum_{i=1}^{n-1}(1/i) + 1/n = \sum_{i=1}^{n}(1/i) \).
Runtime analysis -- idea

\[
\begin{align*}
\text{sum_seq}(n) & \quad | \\
\text{sum_seq}(n-1) & \quad | \\
\text{sum_seq}(n-2) & \quad | \\
\vdots & \quad | \\
\text{sum_seq}(0) & \\
\end{align*}
\]

Each stage takes $O(1)$ time.

$O(n)$ levels of recursion
Run time analysis – Formal Proof

**Theorem:** `sum_seq()` runs in time $O(n)$.

**Proof:** Let $T(n)$= time needed for `sum_seq` to run on $n$. We will show, by induction on $n$, that $T(n) \leq cn+c$, where $c$ is a constant that is the number of operations in one level of `sum_seq()` (i.e. $c$ is the total time needed to test $n==0$?, compute $n-1$, add $1/n$ to the result, etc.)

- **Basis:** for $n=0$, we return directly having spent at most $c$ operations, so $T(0) \leq c$.
- **Induction step (assume $n$, prove $n+1$):** `sum_seq(n+1)` performs at most $c$ operations + calls `sum_seq(n)`, thus:
  $$T(n+1) \leq T(n) + c \leq cn+c + c = c(n+1)+c$$
Warning...

We can not just prove by induction that $T(n)=O(n)$:

$$T(n+1) = T(n)+O(1) = O(n) + O(1) = O(n)$$

Otherwise, even if each level of the recursion took $O(n)$ time then we could *incorrectly* "show" that $T(n)=O(n)$:

$$T(n+1) = T(n)+O(n) = O(n) + O(n) = O(n)$$

The error is that we need the constant $c$ not to change between levels of the recursion.
Theorem: For every non-negative integer $n$, and every real $x$, this function returns $x^n$.

Proof: By induction on $n$:
• Basis: for $n=0$, $\text{power}(x,0)$ returns $1=x^0$.
• Assume $n-1$, prove $n$: $\text{power}(x,n)$ returns $x*\text{power}(x,n-1) = x \cdot x^{n-1} = x^n$

Theorem: Running time is $O(n)$
Proof: $n$ levels of recursion, each taking $O(1)$ time.
Raising to power – doubling

```python
def power(x, n):
    if n==0:
        return 1
    if n%2 == 0:
        t = power(x, n//2)
        return t*t
    return x*power(x, n-1)
```

<table>
<thead>
<tr>
<th>Power(x,*)</th>
<th>Computed by</th>
</tr>
</thead>
<tbody>
<tr>
<td>70</td>
<td>(x^{35})^2</td>
</tr>
<tr>
<td>35</td>
<td>x*x^{34}</td>
</tr>
<tr>
<td>34</td>
<td>(x^{17})^2</td>
</tr>
<tr>
<td>17</td>
<td>x*x^{16}</td>
</tr>
<tr>
<td>16</td>
<td>(x^{8})^2</td>
</tr>
<tr>
<td>8</td>
<td>(x^{4})^2</td>
</tr>
<tr>
<td>4</td>
<td>(x^{2})^2</td>
</tr>
<tr>
<td>2</td>
<td>(x^{1})^2</td>
</tr>
<tr>
<td>1</td>
<td>x*x^{0}</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
**Theorem:** For any $x$ and non-negative integer $n$, this power function returns $x^n$.

**Proof:** by complete induction on $n$.

- **Basis:** For $n=0$, we return 1.

- **Assume for all** $i<n$, prove for $n$:
  - If $n$ is even, $\text{power}(x,n)$ returns $(\text{power}(x, n/2))^2 = (x^{n/2})^2 = x^n$.
  - If $n$ is odd, $\text{power}(x,n)$ returns $x*\text{power}(x,n-1) = x \cdot x^{n-1} = x^n$.

The running time is now $O(\log n)$:

- After 2 levels of recursion $n$ has decreased by a factor of at least two (since either $n$ or $n-1$ is even, in which case the recursive call is with $n/2$)
- Thus we reach $n==0$ after at most $2\log_2 n$ levels of recursion
- Each level still takes $O(1)$ time.
Recursive GCD

def gcd(x, y):
    print("called on", x, y)
    if y == 0:
        return x
    return gcd(y, x % y)
def reverse_user():
    name = input("enter name: ")
    if name != ":
        reverse_user()
    print(name)
def find(x, a):
    find_in_range(x, a, 0, len(a))

def find_in_range(v, a, low, high):
    if high <= low:
        return -1 # not found
    mid = (high+low)//2
    if a[mid] == x:
        return mid
    if a[mid]>x:
        return find(x, a, low, mid)
    return find(x, a, mid+1, high)

Example: find 12 in [1, 3, 5, 9, 11, 12, 15, 17]

<table>
<thead>
<tr>
<th>low</th>
<th>high</th>
<th>mid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>
Run time analysis – $O(\log n)$

Find on length $n$

Find on length $n/2$

Find on length $n/4$

...  

Find on length 1

$O(1)$ work per level

$log n$ levels
Merge-Sort idea

• To sort a list:  
  – Sort the first half of the list \([5, 7, 8]\)  
  – Sort the second half of the list \([1, 3, 6]\)  
  – Merge both lists into a single one \([1, 3, 5, 6, 7, 8]\)
Merge-Sort idea

• To sort a list:
  – Sort the first half of the list
  – Sort the second half of the list
  – Merge both lists into a single one

[5, 7, 8]
[1, 3, 6]
Merge-Sort idea

• To sort a list:
  – Sort the first half of the list
  – Sort the second half of the list
  – Merge both lists into a single one

\[\begin{align*}
1 & 3 & 6 \\
5 & 7 & 8 \\
\end{align*}\]

\[\begin{align*}
1 & [7, 5, 8, 3, 1, 6] \\
[5, 7, 8] & [1, 3, 6] \\
\end{align*}\]
Merge-Sort idea

• To sort a list:
  – Sort the first half of the list
    [7, 5, 8, 3, 1, 6]
  – Sort the second half of the list
    [5, 7, 8]
  – Merge both lists into a single one
    [1, 3, 6]
Merge-Sort idea

• To sort a list:
  – Sort the first half of the list [7, 5, 8, 3, 1, 6]
  – Sort the second half of the list [5, 7, 8]
  – Merge both lists into a single one [1, 3, 6]

1 3 6
5 7 8

-----------------
1 3 5
Merge-Sort idea

• To sort a list:
  – Sort the first half of the list
  – Sort the second half of the list
  – Merge both lists into a single one

\[
\begin{array}{cccc}
1 & 3 & 6 & \square \\
5 & \boxed{7} & 8
\end{array}
\]

\[
\begin{array}{cccc}
1 & 3 & 5 & 6 \\
\end{array}
\]
Merge-Sort idea

• To sort a list:
  – Sort the first half of the list [7, 5, 8, 3, 1, 6]
  – Sort the second half of the list [5, 7, 8]
  – Merge both lists into a single one [1, 3, 6]
**Merge**

**Input:** Two sorted arrays  
**Output:** A single sorted array holding all elements from both lists

def merge(b, c, a):
    i, j, k = 0, 0, 0
    while k<len(a):
        if i < len(b) and (j >= len(c) or b[i]<c[j]):
            a[k] = b[i]
            i = i+1
            k = k+1
        else:
            a[k] = c[j]
            j = j+1
            k = k+1
**Merge**

**Input:** Two sorted arrays  
**Output:** A single sorted array holding all elements from both lists

```python
def merge(b, c, a):
    i, j, k = 0, 0, 0
    while k < len(a):
        if b[i] < c[j]:
            a[k] = b[i]
            i = i + 1
            k = k + 1
        else:
            a[k] = c[j]
            j = j + 1
            k = k + 1
```
def sort(a):
    if len(a) == 1:
        return
    mid = len(a) // 2
    b = a[:mid]
    c = a[mid:]
    sort(b)
    sort(c)
    merge(b, c, a)

>>> a = [5, 7, 2, 3, 8, 0]
>>> sort(a)
    sort called on [5, 7, 2, 3, 8, 0]
        sort called on [5, 7, 2]
            sort called on [5]
            sort called on [7]
        sort called on [2]
    sorted [3, 8, 0] into [3, 8, 0]
    sorted [8, 0] into [8, 0]
    merged [8] and [0] into [0, 8]
    merged [3] and [0, 8] into [0, 3, 8]
    merged [2, 5, 7] and [0, 3, 8] into [0, 2, 3, 5, 7, 8]
    ...
Tree of Calls

\[
\begin{align*}
&n \\
\frac{n}{2} &\quad \frac{n}{2} \\
\frac{n}{4} &\quad \frac{n}{4} &\quad \frac{n}{4} &\quad \frac{n}{4} \\
\vdots &\quad \vdots &\quad \vdots &\quad \vdots \\
1 &\quad 1 &\quad \ldots &\quad 1
\end{align*}
\]
Run time is $O(n \log n)$

- A call to sort(length $k$) requires:
  - $O(k)$ time to split the list / copy it
  - $O(k)$ time to Merge two sub-lists of total length $k$
  - Recursive calls

- The total run has:
  - 1 call on length $n$
  - 2 calls on length $n/2$
  - ...
  - $2^i$ calls on length $n/2^i$
  - ...
  - $n$ calls on length 1

Each level takes $2^i * n/2^i = O(n)$ time

Log($n$) levels
Recursive Definitions

**Fibonacci Numbers:** 1, 1, 2, 3, 5, 8, 13, 21, ...

- \( f_0 = f_1 = 1; \ f_n = f_{n-1} + f_{n-2} \)

**Arithmetic Expressions:** E.g. \( 2 + 3 \times (5 + (3 - 4)) \)

- A number is an expression
- For any expression \( E \): \( (E) \) is an expression
- For any two expressions \( E_1, E_2 \): \( E_1 + E_2, E_1 - E_2, E_1 \times E_2, E_1 / E_2 \) are expressions

**Fractals:**

![Fractal Images](image)

In such cases recursive algorithms are very natural
```python
import turtle

def draw_fractal(length, level):
    if level == 0:
        turtle.forward(length)
    else:
        draw_fractal(length/3, level-1)
        turtle.left(60)
        draw_fractal(length/3, level-1)
        turtle.right(120)
        draw_fractal(length/3, level-1)
        turtle.left(60)
        draw_fractal(length/3, level-1)

for level in range(5):
    turtle.penup()
    turtle.setpos(-300, 300 - 200*level)
    turtle.pendown()
    draw_fractal(500, level)
```
def fib(n):
    if n<=2:
        return 1
    return fib(n-1) + fib(n-2)
Run time analysis

```python
def fib(n):
    if n<=2:
        return 1
    return fib(n-1) + fib(n-2)
```

**Theorem:** Let $T(n)$ = running time on $n$. Then:
$$T(n) = O(\phi^n)$$
where $\phi = (1 + \sqrt{5})/2 = 1.618...$

**Lemma:** For every natural $n$: $T(n) \leq c \cdot (\phi^n - 1)$

**Proof:**
- **Basis:** Take $c$ so that $c \cdot (\phi - 1) >$ number of operation of single level

- **Induction Step:**
  $$T(n) = T(n-1) + T(n-2) + c \leq c \cdot (\phi^{n-1} - 1) + c \cdot (\phi^{n-2} - 1) + c = c \cdot ((\phi + 1) \cdot \phi^{n-2} - 1) = c(\phi^n - 1)$$

  $\phi + 1 = \phi^2$
Memoization

\[ past\_computations = \{1:1, 2:1\} \]

```python
def fib(n):
    if n in past_computations:
        return past_computations[n]
    answer = fib(n-1) + fib(n-2)
    past_computations[n] = answer
    return answer
```

**Theorem:** Runtime is \( O(n) \).

**Proof:** Global analysis: \( O(1) \) work is performed for every entry of past_computations that is filled.
Converting a loop

```python
def iterative_function(inp):
    state = initial_state(inp)
    while cond(state):
        do_something(state)
        state = update(state)
    return answer(state)

def recursive_function(inp):
    recursive_helper(initial_state(inp))

def recursive_helper(state):
    if cond(state):
        do_something(state)
        return recursive_helper(update(state))
    else:
        return answer(state)
```