CSc 120
Introduction to Computer Programming II
Adapted from slides by Dr. Saumya Debray

16: Stacks, Recursion, Search
the runtime stack
How recursion works

```python
>>> def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

>>> fact(4)
24
```
How recursion works

```python
>>> def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n+1)

>>> fact(4)
24
```

We need the value of `n` both before and after the recursive call.

\[ \therefore \text{its value has to be saved somewhere} \]

“somewhere” \(\equiv\) “stack frame”
How recursion works

```python
>>> def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

>>> fact(4)
24
```

Python's runtime system* maintains a stack:
- push a "frame" when a function is called
- pop the frame when the function returns

* "runtime system" = the code that Python executes to make everything work at runtime

"frame" or "stack frame": a data structure that keeps track of variables in the function body, and their values, between the call to the function and its return
How recursion works

```python
>>> def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

>>> fact(4)
24
```

Python's runtime system* maintains a **stack**:
- push a "frame" when a function is called
- pop the frame when the function returns

* "runtime system" = the code that Python executes to make everything work at runtime
How recursion works

```python
>>> def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

>>> fact(4)
24
```

![Diagram of stack frame for fact()]
How recursion works

```python
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

>>> fact(4)
24
```

Python code showing the recursive function `fact(n)` and its execution for `n = 4`, resulting in 24. The diagram illustrates stack frames and return values for each recursive call, with the runtime stack and stack frames depicted.
How recursion works

```python
>>> def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

>>> fact(4)
24
```
How recursion works

```python
>>> def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

>>> fact(4)
24
```

![Diagram of stack frame for fact()](image)
How recursion works

```python
>>> def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

```python
>>> fact(4)
24
```

Stack frame for `fact()`

Stack top

<table>
<thead>
<tr>
<th>fact(1)</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>fact(2)</td>
<td>2</td>
</tr>
<tr>
<td>fact(3)</td>
<td>3</td>
</tr>
<tr>
<td>fact(4)</td>
<td>4</td>
</tr>
</tbody>
</table>

Runtime stack
How recursion works

```python
>>> def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

>>> fact(4)
24
```
How recursion works

```python
>>> def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

>>> fact(4)
24
```
How recursion works

```python
>>> def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

>>> fact(4)
24
```
How recursion works

```python
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

>>> fact(4)
24
```

Stack frame for `fact()`

Value of `n`

Value from recursive call

Return value

Runtime stack

```plaintext
fact(0)

fact(1)

fact(2)

fact(3)

fact(4)

0 1 1 1

2 1 2

3

4
```
How recursion works

```python
>>> def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

>>> fact(4)
24
```

![Diagram of recursion stack](image)
How recursion works

```python
>>> def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

>>> fact(4)
24
```

The code defines a function `fact` that calculates the factorial of a number recursively. The function has a stack frame for each call, and values are pushed onto the runtime stack for each recursive call. The values are then popped off the stack in reverse order as the recursion unwinds, calculating the factorial. The stack frame for `fact()` includes the value of `n`, the value from the recursive call, and the return value.
How recursion works

```python
>>> def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)
```

```
>>> fact(4)
24
```

---

Stack frame for `fact()`

Value of `n`

Value from recursive call

Return value

---

Runtime stack

```
fact(0)  0  1
```

```
fact(1)  1  1  2
```

```
fact(2)  2  1  2
```

```
fact(3)  3  2  6
```

```
fact(4)  4  6  24
```

Stack top
The runtime stack

• The use of a *runtime stack* containing *stack frames* is not specific to recursion
  – all function and method invocations use this mechanism
  – not just in Python, but other languages as well (Java, C, C++, ...)

```python
>>> def g(L, val):
    L.append(val)

>>> def f(n):
    X = [n-1]
    g(X, n)
    X.append(n+1)
    print(X)

>>> f(10)
[9, 10, 11]
```
The runtime stack

```python
>>> def g(L, val):
    L.append(val)
```

```python
>>> def f(n):
    X = [n-1]
    g(X, n)
    X.append(n+1)
    print(X)
```

```python
>>> f(10)
[9, 10, 11]
```
The runtime stack

```python
>>> def g(L, val):
    L.append(val)

>>> def f(n):
    X = [n-1]
    g(X, n)
    X.append(n+1)
    print(X)

>>> f(10)
[9, 10, 11]
```

**f's stack frame**

```
x  [9]
```

**runtime stack**

```
n  10
```
The runtime stack

```python
>>> def g(L, val):
    L.append(val)

>>> def f(n):
    X = [n-1]
    g(X, n)
    X.append(n+1)
    print(X)

>>> f(10)
[9, 10, 11]
```

Argument passing: the callee is passed a reference to the argument object.
The runtime stack

```python
>>> def g(L, val):
    L.append(val)

>>> def f(n):
    X = [n-1]
    g(X, n)
    X.append(n+1)
    print(X)

>>> f(10)
[9, 10, 11]
```

Argument passing: the callee is passed a reference to the argument object

⇒ the change made in g() is visible in f()
The runtime stack

```python
>>> def g(L, val):
    L.append(val)

>>> def f(n):
    X = [n-1]
    g(X, n)
    X.append(n+1)
    print(X)

>>> f(10)
[9, 10, 11]
```
The runtime stack

```python
>>> def g(L, val):
    L.append(val)

>>> def f(n):
    X = [n-1]
    g(X, n)
    X.append(n+1)
    print(X)

>>> f(10)
[9, 10, 11]
```
The runtime stack: summary

• **Runtime stack**: holds information about function (and method) activations

• **Stack frame** for a function:
  – holds information about the variables in the function body
  – pushed when the function is called
  – popped when it returns

• **Argument passing**: a reference to the argument value (an object) is passed
  – if the value is mutable, changes made by the callee are visible in the caller
search revisited
Game tree search revisited

Recall our tic-tac-toe program

Given a starting position,

- it generates successive positions from different possible moves
- evaluates the effect of continuing play from each of these positions
- picks a move that leads to the best position after some number of turns $n$ ($n = \text{“lookahead”}$)
Game tree search revisited

Recall our tic-tac-toe program

Given a starting position,

- it generates successive positions from different possible moves
- evaluates the effect of continuing play from each of these positions
- picks a move that leads to the best position after some number of turns
Game tree search revisited

def eval_pos(pos, turn):
    if game_over(pos):
        return win_or_loss(pos, turn)
    else:
        while next_pos != None:
            next_pos = generate_next_pos(pos, turn)
            if next_pos != None:
                result = eval_pos(next_pos, next[turn])
    ...

def generate_next_pos(pos, turn):
    for i in range(3):
        for j in range(3):
            if pos[i][j] == ' ':
                pos[i][j] = turn
                return pos
    return None
**Game tree search revisited**

```python
def eval_pos(pos, turn):
    if game_over(pos):
        return win_or_loss(pos, turn)
    else:
        while next_pos != None:
            next_pos = generate_next_pos(pos, turn)
            if next_pos != None:
                result = eval_pos(next_pos, next[turn])

def generate_next_pos(pos, turn):
    for i in range(3):
        for j in range(3):
            if pos[i][j] == ' ':
                pos[i][j] = turn
                return pos
    return None
```

updates the position
Game tree search revisited

Because arguments are passed by object reference:

- changes made to the board here will be visible here

- but these board positions are supposed to be independent!
Game tree search revisited

Solution: create a copy of the board position

A refresher on copying:

**without copying**

```python
>>> x = [[1, 2, 3], [4, 5, 6]]
>>> y = x
>>> y[0].append(73)
>>> x
[[1, 2, 3, 73], [4, 5, 6]]
```

**with deep copying**

```python
>>> from copy import *
>>> x = [[1, 2, 3], [4, 5, 6]]
>>> y = deepcopy(x)
>>> y[0].append(73)
>>> y
[[1, 2, 3, 73], [4, 5, 6]]
>>> x
[[1, 2, 3], [4, 5, 6]]
```
Game tree search revisited

Solution: create a copy of the board position

```python
from copy import *
...

def generate_next_pos(pos, turn):
    new_pos = deepcopy(pos)
    for i in range(3):
        for j in range(3):
            if new_pos[i][j] == ' ':
                new_pos[i][j] = turn
    return new_pos

return None
```

updates to new_pos don't change pos
search problems: examples
Word morph

• Change one word into another by changing one letter at a time

Examples:
  – cat → cot → cog → dog
  – head → heal → hell → hall → tall → tail

• Also called Word Ladder
Word morph

• Change one word into another by changing one letter at a time

Examples:
- cat → cot → cog → dog
- head → heal → hell → hall → tall → tail

• Imagine a tree where each level is the set of possible words created by changing one character
Word morph: sample tree

one letter different from prior level
Word morph

• Change one word into another by changing one letter at a time

• All of the words generated by changing one letter go in the next level of the tree: head

hea_  he_d  h_ad  _ead

  heal  held  hoed  mead
heap  bead
heat  lead
...
...
Word morph

• Given a dictionary of valid words
  – Generate the set of words that differ from a word $w1$ by one letter

• Solution 1
  – For each position $i$ in $w1$,
    for each letter in the alphabet,
    create a new word by changing position $i$ to the next letter in the alphabet
    if it's in the dictionary, add it to the set of words*

*unless it's been seen already
Word morph

• Given a dictionary of valid words
  – Generate the set of words that differ from a word w1 by one letter

• Solution 2
  – Write a distance function that computes the number of positions in two strings where the two strings differ
    o distance(heap, heat) returns 1
    o distance(keep, beep) returns 2
  – For each word w2 in the dictionary
    if the distance between w1 and w2 is 1, then add w2 to the set of words
    *unless it's been seen (or check for that)
Exercise

Write a function `dist(w1, w2)` that returns the number of positions where words `w1` and `w2` differ. It requires that `len(w1) == len(w2).

Use an assert to verify the lengths of `w1` and `w2` are the same.

Use a list comprehension in your function.
Word morph: sample tree

• How do we generate and search the possibilities?
def main():
    (word1, word2, word_list) = read_input()
    morph_seq = morph(list(word1), list(word2), word_list, [])
    print_seq(morph_seq)
def morph(w1, w2, word_list, Seen):
    if w1 == w2:
        return [w2]
    elif w1 in Seen:
        return []
    else:
        candidate_list = next_words(w1, w2, word_list):
        for candidate in candidate_list:
            next = update_word(w1, candidate)
            result = morph(next, w2, word_list, Seen + [w1])
            if result != []:
                return [w1] + result
        return []

generate the list of words to try next
def morph(w1, w2, word_list, Seen):
    if w1 == w2:
        return [w2]
    elif w1 in Seen:
        return []
    else:
        candidate_list = next_words(w1, w2, word_list):
        for candidate in candidate_list:
            next = update_word(w1, candidate)
            result = morph(next, w2, word_list, Seen + [w1])
            if result != []: return [w1] + result
        return []

update the “current position”
def morph(w1, w2, word_list, Seen):
    if w1 == w2:
        return [w2]
    elif w1 in Seen:
        return []
    else:
        candidate_list = next_words(w1, w2, word_list):
        for candidate in candidate_list:
            next = update_word(w1, candidate)
            result = morph(next, w2, word_list, Seen + [w1])
            if result != []:
                return [w1] + result
        return []
Word morph: search

```python
def morph(w1, w2, word_list, Seen):
    if w1 == w2:
        return [w2]
    elif w1 in Seen:
        return []
    else:
        candidate_list = next_words(w1, w2, word_list):
        for candidate in candidate_list:
            next = update_word(w1, candidate)
            result = morph(next, w2, word_list, Seen + [w1])
            if result != []:  
                return [w1] + result
        return []
```

if a solution is found, return immediately. Otherwise, keep searching (i.e., iterating).
def next_words(wd1, wd2, word_list):
    cands = [wd for wd in word_list \\
             if len(wd) == len(wd1) and dist(wd, wd1) == 1]
    cands.sort(key = lambda wd : dist(wd, wd2))

# dist(w1, w2) returns the number of positions where words
# w1 and w2 differ. It requires that len(w1) == len(w2).

def dist(w1, w2):
    assert len(w1) == len(w2)
    diffs = [i for i in range(len(w1)) if w1[i] != w2[i]]
    return len(diffs)
# File: “morph.py”
# Author: Saumya Debray
import sys
from copy import *
DICT = 'WORDS.txt'
def read_input():
    # read the dictionary into a list
    try:
        dict_file = open(DICT)
    except IOError:
        print('ERROR: could not open file: ' + dicsilename)
sys.exit(1)

    word_list = []
    for word in dict_file:
        word_list.append(word.strip())
    # read the two words to be morphed
    word1 = input('Word 1: ')
    word2 = input('Word 2: ')
    return (word1, word2, word_list)

# dist(w1, w2) returns the no. of positions where w1, w2 differ.
def dist(w1, w2):
    assert len(w1) == len(w2)
    diffs = [i for i in range(len(w1)) if w1[i] != w2[i]]
    return len(diffs)
def morph(w1, w2, word_list, Seen):
    if w1 == w2:
        return [w2]
    elif w1 in Seen:
        return []
    else:
        candidates = [w for w in word_list \ 
                      if len(w) == len(w1) and dist(w, w1) == 1]
        # consider candidates closer to w2 first
        candidates.sort(key = lambda w:dist(w, w2))
        for cand in candidates:
            result = morph(cand, w2, word_list, Seen + [w1])
            # a non-empty result means a successful morph
            if result != []:
                return [w1] + result
        return []
def print_seq(word_list):
    if word_list == []:
        print('Sorry, no morph sequence found')
    else:
        out_str = ' --> '.join(word_list)
        print(out_str)
def main():
    (word1, word2, word_list) = read_input()
    morph_seq = morph(word1, word2, word_list, [])
    print_seq(morph_seq)
main()
Word morph: example runs

• cat → dog
  – cat, cot, cog, dog

• head → tail
  – head, heal, heel, hell, hall, tall, tail

• nose → chin
  – nose, Bose, dose, dole, dale, dame, came, cage, cake, cape, care, card, carp, camp, lamp, lame, fame, fare, dare, darn, damn, dawn, down, gown, sown, soon, coon, coin, chin

why the extra words? 😕
Word morph: example runs

depth-first search!
Word morph: example runs

breadth-first search
lambda expressions

• Anonymous functions can be made with the lambda keyword
  – `lambda a, b : a + b`

• This is a syntactic short hand

• Restricted to single expressions

• Lambda functions can be used wherever a function object is valid
lambda expressions

• Example:
  >>> def make_incrementer(n):
        return lambda x: x + n

  >>> f = make_incrementer(10)
  >>> f
  <function make_incrementer.<locals>.<lambda> at 0x10570bf28>
  >>> f(2)
  12
  >>> f(100)
  110
  >>>
Challenge

• This version of the word morph game works with just one single word
• What would it take to let the program work with more than one word?
  – keep total length the same

  e.g.: software → soft are → soft ear

• Looking for:

  Wildcats → Beat ASU
The Cracker Barrel peg game

Repeatedly jump over and remove pegs until there is just one peg left on the board
The Cracker Barrel peg game
The Cracker Barrel peg game

# the recursive search

def solve(board, npins, movelist):
    if npins == 1:  # success!
        print_moves(movelist)
    else:
        mvs = [mv for mv in moves if legal_move(board, mv)]
        for mv in mvs:
            newboard = update_board(board, mv)
            solve(newboard, npins-1, movelist+[mv])
# the recursive search

def solve(board, npins, movelist):
    if npins == 1:    # success!
        print_moves(movelist)
    else:
        mvs = [mv for mv in moves if legal_move(board, mv)]
        for mv in mvs:
            newboard = update_board(board, mv)
            solve(newboard, npins-1, movelist+[mv])

The Cracker Barrel peg game
The Cracker Barrel peg game

# moves is a list of the possible moves. An entry (src,mid,dst) 
# indicates a move from src to dst jumping over mid.

moves = [(0,1,3),(0,2,5),(1,3,6),(1,4,8),
         (2,4,7),(2,5,9),(3,1,0),(3,4,5),(3,6,10),
         (3,7,12),(4,7,11),(4,8,13),(5,2,0),
         (5,4,3), (5,8,12),(5,9,14),(6,3,1),
         (6,7,8),(7,4,2),(7,8,9),(8,4,1),(8,7,6),
         (9,5,2),(9,8,7),(10,6,3),(10,11,12),
         (11,7,4),(11,12,13),(12,7,3),(12,8,5),
         (12,11,10),(12,13,14),(13,8,4),
         (13,12,11),(14,9,5),(14,13,12)]
The Cracker Barrel peg game

# The board is a dictionary \{k_1:v_1, ..., k_n:v_n\} where the \( k_i \) are # positions and the \( v_i \) are 0 or 1 indicating whether the # position is occupied.

def update_board(board, move):
    (src, mid, dst) = move
    newboard = copy(board)  # create a (shallow) copy of the board
    newboard[src] = 0       # pin moved away from this position
    newboard[mid] = 0       # pin removed from this position
    newboard[dst] = 1       # pin lands on this position
    return newboard
The Cracker Barrel peg game

def main():
    if len(sys.argv) < 2:
        print("Usage: crackerbarrel.py config_string")
        sys.exit(1)

    # "011111111111111" --> 29,760 solutions
    initial_config = sys.argv[1]
    board = build_board(initial_config)
    npins = initial_config.count('1')
    solve(board, npins, [])
The Cracker Barrel peg game:

code

# File: crackerbarrel.py
# Author: Saumya Debray

import sys
from copy import *

""" This program computes all solutions the "Cracker-Barrel Problem,
ignoring symmetries. """

# moves is a list of the possible moves. An entry (src,mid,dst)
# a move from src to dst jumping over mid.

moves = [(0,1,3),(0,2,5),(1,3,6),(1,4,8),(2,4,7),(2,5,9),
(3,1,0),(3,4,5),(3,6,10),(3,7,12),(4,7,11),(4,8,13),
(5,2,0),(5,4,3),(5,8,12),(5,9,14),(6,3,1),(6,7,8),
(7,4,2),(7,8,9),(8,4,1),(8,7,6),(9,5,2),(9,8,7),
(10,6,3),(10,11,12),(11,7,4),(11,12,13),
(12,7,3),(12,8,5),(12,11,10),(12,13,14),
(13,8,4),(13,12,11),(14,9,5),(14,13,12)]

# build_board(config) returns a board corresponding to the
# configuration string config. The board is a dictionary
# {k:v1, ..., kn:vn} where the ki are positions and the vi are 0 or 1
# indicating whether that position is occupied.

def build_board(config):
    n = len(config)
    occupancy_list = [int(k) for k in list(config)]
    return dict(zip(range(n), occupancy_list))

# update_board(board, (src,mid,dst)) returns a new board that gives
# the result of making a move (src,mid,dst), i.e., moving a pin from
# position src to position dst and thereby removing the pin at position
# mid, in the given board.

def update_board(board, move):
    (src,mid,dst) = move
    newboard = copy(board)
    newboard[src] = 0
    newboard[mid] = 0
    newboard[dst] = 1
    return newboard

# legal_move(board, (src,mid,dst)) returns True if (src,mid,dst) is a
# legal move in the given board; False otherwise.

def legal_move(board, mov):
    (src,mid,dst) = mov
    return board[src] == 1 and board[mid] == 1 and board[dst] == 0

# solve() performs a brute-force exploration of the search space.

def solve(board, npins, movelist):
    if npins == 1:    # success!
        print_moves(movelist)
    else:
        mvs = [mv for mv in moves if legal_move(board, mv)]
        for mv in mvs:
            newboard = update_board(board, mv)
            solve(newboard, npins-1, movelist+[mv])

def print_moves(L):
    line = ""
    for i in range(len(L)):
        (src,mid,dst) = L[i]
        line = line + "[" + str(src) + "->" + str(dst) + "]"
    print(line)

def main():
    if len(sys.argv) < 2:
        print("Usage: crackerbarrel.py config_string")
        sys.exit(1)
    iniEal_config = sys.argv[1]
    board = build_board(iniEal_config)
    npins = iniEal_config.count('1')
    solve(board, npins, [])

main()
The Cracker Barrel peg game

# moves is a list of the possible moves. An entry (src,mid,dst) # indicates a move from src to dst jumping over mid.

moves = [(0,1,3),(0,2,5),(1,3,6),(1,4,8),
   (2,4,7),(2,5,9),(3,1,0),(3,4,5),(3,6,10),
   (3,7,12),(4,7,11),(4,8,13),(5,2,0),
   (5,4,3), (5,8,12),(5,9,14),(6,3,1),
   (6,7,8),(7,4,2),(7,8,9),(8,4,1),(8,7,6),
   (9,5,2),(9,8,7),(10,6,3),(10,11,12),
   (11,7,4),(11,12,13),(12,7,3),(12,8,5),
   (12,11,10),(12,13,14),(13,8,4),
   (13,12,11),(14,9,5),(14,13,12)]

• This formulation of the set of possible moves is specific to a game with 5 rows of pins
• What would a generalization to n rows look like?