De-recursion

Many times, recursion gives us a clean way to think about problems and solve them.

But a recursive program is often slower than non recursive version.

So sometimes, after finding a recursive solution, we want to transform it to a non recursive solution.

Understanding how the non recursive function also helps us understand the recursive version better.

Example: Binary Search

Recall the recursive code for binary search:

```python
In [1]: def bin_search(L,item):
    n = len(L)
    if n==0:
        return -1
    m = int(n/2)
    if L[m]==item:
        return m
    if L[m]>item:
        return bin_search(L[:m],item)
    res = bin_search(L[m+1:n],item)
    return -1 if res== -1 else m+1+res
```

To make it non recursive we will do the following:

```python
In [2]: def bin_search_nr(L,item):
    left = 0
    right= len(L)
    while right-left > 0:
        m = int((left+right)/2)
        if L[m]==item:
            return m
        if L[m]>item:
            right = m
        else:
            left = m+1
    return -1
```
Example 2: Selection sort

```python
In [6]: def find_min_index(L):
    ...:     current_index = 0
    ...:     current_min = L[0]
    ...:     for j in range(1, len(L)):
    ...:         if current_min > L[j]:
    ...:             current_min = L[j]
    ...:             current_index = j
    ...:     return current_index

In [7]: def selection_sort(L):
    ...:     if len(L) <= 1:
    ...:         return L # a one-element list is always sorted
    ...:     min_idx = find_min_index(L) # non-recursive helper function
    ...:     L[0], L[min_idx] = L[min_idx], L[0]
    ...:     return L[0] + sort(L[1:len(L)])

In [8]: def selection_sort_nr(L):
    ...:     for i in range(len(L)):
    ...:         min_idx = i + find_min_index(L[i:])
    ...:         L[i], L[min_idx] = L[min_idx], L[i]
    ...:     return L

In [9]: selection_sort_nr([3, 1, 4, 1, 5, 9, 2])

Out[9]: [1, 1, 2, 3, 4, 5, 9]
```

Example 3: Merge sort
In [10]: `def` `merge_lists(L1,L2):`
   `i=0`
   `j=0`
   `res = []`
   `while i<len(L1) and j<len(L2):`
      `if L1[i] < L2[j]:`
         `res.append(L1[i])`
         `i += 1`
      `else:`
         `res.append(L2[j])`
         `j += 1`
   `res += L1[i:]+L2[j:]`
   `return res`

In [11]: `def` `merge_sort(L):`
   `if len(L) <= 1:`
      `return L`
   `m = int(len(L)/2)`
   `L1 = merge_sort(L[0:m])`
   `L2 = merge_sort(L[m:])`
   `return merge_lists(L1,L2)`

In [12]: `merge_sort([3,1,4,1,5,9,2])`
Out[12]: `[1, 1, 2, 3, 4, 5, 9]`

In [13]: `def` `merge_sort_nr(L):`
   `lists = [ [x] for x in L]`
   `while len(lists)>1:`
      `new_lists = []`
      `if len(lists) % 2:`
         `lists.append([])`
      `for i in range(0,len(lists)-1,2):`
         `new_lists.append(merge_lists(lists[i],lists[i+1]))`
      `lists = new_lists`
   `return lists[0]`

In [14]: `merge_sort_nr([3,1,4,1,5,9,2])`
Out[14]: `[1, 1, 2, 3, 4, 5, 9]`

**Graphs**
Often in computation we have data from the world, and a question we want to answer about these data.

To do so, we need to find a model for the data, and a way to translate our question into a mathematical question about the model.

Here are some examples:

- Suppose you have a map of Addis Ababa and want to find out what's the fastest way to get from the national museum to the market.
- Suppose you are Facebook and you are trying to figure out how many friends of friends does the average Ethiopian has.
- Suppose you are a geneticist, and are trying to figure out which genes are related to a particular type of colon cancer.

What is perhaps most surprising is that these and any many other questions, all use the same mathematical model of a graph.
A graph is just a way to store connections between pairs of entities:

- The graph of Addis’s roads could be composed of all street intersections, with a connection between intersection $u$ and intersection $v$ if they are directly connected by a road.
- The Facebook graphs is composed of all Facebook users, with a connection between user $u$ and user $v$ if they are friends.
- The gene-symptom interaction graph is composed of all genes and all "symptoms" (also known as phenotypes: some observable differences in people), where gene $u$ is connected to symptom $v$ if there is a correlation between people having the gene $u$ and symptom $v$.

Mathematically, a graph is a set $V$ of vertices and a set $E$ of pairs of these vertices which is known as the set of edges. We say that a vertex $u \in V$ is connected to $v \in V$ if the pair $(u, v)$ is in $E$.

A graph where $(u, v) \in E$ if and only if $(v, u) \in E$ is known as an undirected graphs. Undirected graphs form an important special case, and we will mostly be interested in those graphs.

Sometimes the edges (or vertices) of the graph are labeled (often by a number), for example in the case of the road network, we might label every road segment with the average time it takes to travel from one end to the other.

There are two main representations for graphs. We can always assume the vertices are simply identified by the numbers 1 to $n$ for some $n$.

The adjacency list representation is an array $L$ where $L[i]$ is the list of all neighbors of the vertex $i$ (i.e., all $j$ such that $(i, j) \in E$)

The adjacency matrix representation is an $n \times n$ two-dimensional array $M$ (i.e., matrix) such that $M[i][j]$ equals 1 if $j$ is a neighbor of $i$ and equals 0 otherwise.

Questions

- If a graph has $n$ vertices and $m$ edges - how big is its adjacency list representation? how big is its adjacency matrix representation?

- Given a graph $G$ on $n$ vertices and two vertices $i, j$, how long can it take us (in the worst case) to find out if $j$ is a neighbor of $i$ when $G$ is represented in the adjacency list form? How long will it take in the adjacency matrix form?
Examples:

In [16]: G = [[1],[2],[3],[4]]

In [17]: draw_graph(G)

deck: 0

In [21]: G = [[1,2,3,4,5,6]]

deck: 0
In [23]:
    n = 20
    G = [ [(i+1) % n] for i in range(n) ]
    draw_graph(G)

    spectral_layout

In [24]:
    def grid_neighbors(i,j,n):
        if i==n-1 and j== n-1: return []
        if i==n-1:
            return [i*n+j+1]
        if j==n-1:
            return [(i+1)*n+j]
        return [n*i+((j+1) % n), n*((i+1) % n)+j]

In [25]:
    n = 5
    G = [ grid_neighbors(i,j,n) for i in range(n) for j in range(n) ]
**Graph connectivity**

Given $i$, $j$ and a graph $G$: find out if $j$ is connected to $i$ (perhaps indirectly) in the graph.

Here is a natural suggestion for a recursive algorithm:

$\text{connected}(i, j, G)$ is True if $i$ is a neighbor of $j$, and otherwise it is True if there is some neighbor $k$ of $i$ such that $k$ is connected to $j$.

Let's code it up try to see what happens:

```python
In [27]:
    def connected(i, j, G):
        sys.stdout.write('.
')
        if j in G[i]:
            return True
        return any([connected(k, j, G) for k in G[i]])

In [28]:
    def undir(G):
        n = max(max(G[i]) if G[i] else 0 for i in range(len(G)))
        n = max(n+1, len(G))
        _G = [[] for i in range(n)]
        for i in range(len(G)):
            for j in G[i]:
                if not j in _G[i]:
                    _G[i].append(j)
                    if not i in _G[j]:
                        _G[j].append(i)
        return _G
```
In [29]:
G = [[1],[2],[3],[4],[]]
draw_graph(G)
spectral_layout

In [30]:
G = undir(G)
G
Out[30]:
[[1], [0, 2], [1, 3], [2, 4], [3]]

In [31]:
connected(0,1,G)
.
Out[31]:
True

In [32]:
connected(0,2,G)
.
Out[32]:
True
The problem is that we are getting into an infinite loop! We can fix this by remembering which vertices we visited.

In [34]: def grid_input(n): # return a n by n grid with an isolated vertex
   G = [ [ grid_neighbors(i,j,n) for i in range(n) for j in range(n) ]
   G.append([])
   G = undir(G)
   return (0,len(G)-1,G)

In [35]: def connected(source,target,G):
   added = [False for i in range(len(G))]
   added[source] = True
   to_visit = [source] # to visit: list of vertices that are definitely connected to the source
   while to_visit:
       step_pc() # count how many times the while loop is executed
       i = to_visit.pop()
       if i==target:
           return True
       for j in G[i]:
           if not added[j]:
               added[j] = True
               to_visit.append(j)
   return False
In [36]: G = undir([[1],[2],[0],[3]])
    : draw_graph(G)
    : spring100_layout

In [37]: print connected(0,1,G) , connected(0,3,G)

True False
In [24]: # running time of connectivity algorithm

0.684 micro-seconds per step
(array([4], dtype=int64),)
Curve (steps): $n$

Let's see how the evolution of the algorithm looks on a typical graph:
In [38]: def connected_viz(source, target, G, layout_method=None):
    initialize_animation(G, my_layout_method=layout_method)
    visited = [False for i in range(len(G))]
    to_visit = [source]  # to visit: list of vertices that are definitely connected to the source
    while to_visit:
        step_pc()  # count how many times the while loop is executed
        i = to_visit.pop()
        color(i, 'r')  # red: observed
        if i == target:
            return True
        visited[i] = True
        for j in G[i]:
            if not visited[j]:
                to_visit.append(j)
                color(j, 'g')  # green: waiting to be visited
    return False

In [41]: (s, t, G) = grid_input(5)

In [42]: draw_graph(G, 'grid_layout')

grid_layout

![Graph Visualization](image)
In [43]: `connected_viz(s,t,G,'grid_layout')`

Out[43]: False
In [44]: show_animation()
    
saving..
    rendering..

Out[44]:

LIFO vs FIFO
In [45]:

    def connected_FIFO(source, target, G):
        added = [False for i in range(len(G))]
        added[source] = True
        to_visit = [source] # to visit: list of vertices that are definitely connected to the source
        while to_visit:
            i = to_visit.pop(0) # remove first element
            if i==target:
                return True
            for j in G[i]:
                if not added[j]:
                    added[j] = True
                    to_visit.append(j)
        return False

In [46]:

    def connected_FIFO_viz(source, target, G, layout_method = None):
        initialize_animation(G, my_layout_method=layout_method)
        added = [False for i in range(len(G))]
        added[source] = True
        to_visit = [source] # to visit: list of vertices that are definitely connected to the source
        while to_visit:
            step_pc() # count how many times the while loop is executed
            i = to_visit.pop(0) # remove first element
            color(i,'r') # red: observed
            if i==target:
                return True
            for j in G[i]:
                if not added[j]:
                    added[j] = True
                    to_visit.append(j)
                    color(j,'g') # green: added to queue
        return False
The function `connected` is known as **depth first search** and `connected_FIFO` is known as **breadth first search**

**Wrapping up**

This week you actually managed to do some pretty impressive work - **congratulations**
What I hope you learned:

- Coding is about **understanding what problem you need to solve** then **breaking it into smaller problems**
- This is not about typing or computers but about **thinking**, just like math.

My main hope:

- This got you excited about learning more about computer science.

**Ask me anything.**

- About computer science
- About Harvard
- About studying in the U.S.
- Anything else

**Thank you for a great week!!**

You are always welcome to contact me:

Email: b@boazbarak.org

Web page: http://www.boazbarak.org

In [ ]: