Today:

- GUI
  - Packing
  - Events
  - Canvas
  - OptionMenu

- Optimization
  - Traveling Sales Man

- Questions From Previous Exams
In Python, we interact with the GUI using widgets.

Widgets are placed inside other widgets.

The base widget is the root, initialized by Tk().

The GUI starts “running” when we call root.mainloop().
Packing

- packing is a simple way to put widgets in geometric order
- packing only assigns by top, bottom, left, right
use frames in order to sub-divide a screen
Some widgets (such as Button) can have a command associated with them.

General interactions with a widget (such as mouse over, mouse clicks, etc) can have a handler function *bound* to them.

```python
def button_click():
    print("Left click on button!")

def right_button_click(event):
    print("Right click on button at ", event.x, event.y)

root = tk.Tk()
button = tk.Button(root, text='A Button', command=button_click)
button.bind('<Button-3>', right_button_click)
```

The handler function is automatically passed an *event* object.
We can also define what happens when we interact with the GUI *window*. This is done similarly to *bind*, using the *protocol* method. A useful protocol is *WM_DELETE_WINDOW*, for when the window ‘x’ button is pressed.

```python
def on_close():
    if input('are you sure?') == 'y':
        root.destroy()

root = tk.Tk()
root.protocol("WM_DELETE_WINDOW", on_close)
root.mainloop()
```

We need to explicitly destroy the window!
Another way to generate an event is to schedule it using `after`.

Using `after` can also be used to generate sub-loops of mainloop!

```python
def scheduled():
    print("About time!"
            root.after(1000, scheduled)

root = tk.Tk()
root.after(1000, scheduled)
root.mainloop()
```
The Canvas Widget

- The Canvas widget provides an area on which things can be displayed ("drawn")
- Drawing is done via different `create` methods
- The create method returns an item id

```python
root = tk.Tk()
c = tk.Canvas(root, bg='white', height=100, width=100)
c.pack()
c.create_rectangle((50, 50), (100, 100), fill='blue')
c.create_rectangle(0, 0, 50, 50, fill='yellow', outline='blue')
c.create_oval(50, 0, 100, 50, fill='red')
root.mainloop()
```
The OptionMenu Widget

- Allows a selection from multiple options
- Takes a `VariableClass` object and options for it
- A `VariableClass` state can be queried by it’s `get()` method

```python
root = tk.Tk()
choice = tk.StringVar()
choice.set("Default")
options = tk.OptionMenu(root, choice, 'option1', 'option2', 'option3')
options.pack()
root.mainloop()
```
Usfull Links

- Events, Binding, and Protocol Handlers

- Packing
  http://effbot.org/tkinterbook/pack.htm

- Tkinter widgets
  http://effbot.org/tkinterbook/tkinter-index.htm#class-reference

- A couple of tutorials
  http://www.python-course.eu/python=tkinter.php
  http://zetcode.com/gui/tkinter/
Optimization
Optimization Problems

- A set of possible solutions $U$
- A value function $V: U \rightarrow \mathbb{R}$
- The problem: Find a solution $x \in U$ such that $V(x)$ is maximal/minimal
Optimization Problems

- Most questions you can think of can be phrased as an optimization problem!

- Every problem in which we need to find some solution can be phrased as an optimization problem
  - Soduko
  - 8 Queens
Some optimization problems have an efficient algorithms to solve them

But in many cases, finding an optimal solution is computationally hard

Some times we are fine with just finding an approximated solution to the problem

More about this in the *Algorithms* course!
The Traveling Salesman Problem

- There are N different cities
- The cities are connected by a network of roads
- Each road takes a different time to travel
- A salesman wishes to travel through all the cities
- The salesman begins at a specific city and has to finish there also
- The problem: finding the fastest route
TSP as an optimization problem

- What is the set of solutions?
- What is the value function?

The problem can be represented using a graph:

- Each city is a vertex ($|V| = N$)
- Each road between two cities is assigned an edge
- Each edge is assigned the time it takes to travel the respective road
TSP as an optimization problem

- TSP is a hard problem!
- Efficient approximation algorithms exist
- However, finding the shortest path between 2 nodes in a graph is possible in $O(|V|^2)$
TSP as an optimization problem

- We can assume our graph is complete (every two vertices are connected by an edge)
- We can also assume each edge is assigned the minimal time it takes to travel between the cities it connects
- How can we solve it?
TSP Greedy Algorithm

- We could try a greedy algorithm
  - Move to the nearest city in each step
  - But as we already know – the result might not be optimal
We could a *local search*:

For a given permutation \( P \), define a *neighbor* permutation as a permutation where we switched the order of 2 cities in \( P \)
We could a local search:

- For a given permutation $P$, define a *neighbor* permutation as a permutation where we switched the order of 2 cities in $P$.
- Start with a random permutation.
- Each step - move to a neighbor permutation with a better value (while there is one).
- Local search will result in a *local optimum*!
We could use brute force recursion!

Actually very similar to best known algorithm

Algorithm idea:

- Recursively try each permutation of city ordering
- Each branch of the recursion tree returns the best result it found
class GraphNode():
    def __init__(self, name):
        self.name = name
        self.neighbors = {}

    def add_neighbor(self, neighbor, distance):
        if neighbor not in self.neighbors or 
        self.neighbors[neighbor] > distance:
            self.neighbors[neighbor] = distance

    def get_distance(self, neighbor):
        if neighbor in self.neighbors:
            return self.neighbors[neighbor]

    def get_neighbors(self):
        return self.neighbors

    def get_name(self):
        return self.name
CompleteGraph

class CompleteGraph():
    def __init__(self, initial_vertex):
        self.vertices = set()
        self.vertices.add(initial_vertex)
        self.max_distance = 0

    def add_vertex(self, vertex, neighbor, distance):
        self.vertices.add(vertex)
        vertex.add_neighbor(neighbor, distance)
        neighbor.add_neighbor(vertex, distance)
        self.max_distance += 2*distance

    def add_edge(self, vertex1, vertex2, distance):
        if vertex1 in self.vertices and vertex2 in self.vertices:
            vertex1.add_neighbor(vertex2, distance)
            vertex2.add_neighbor(vertex1, distance)
CompleteGraph (cont.)

```python
def update_distances(self):
    for vertex1 in self.vertices:
        for vertex2 in self.vertices:
            min_dist = self.find_min_path(vertex1, vertex2)
            self.add_edge(vertex1, vertex2, min_dist)

def get_max_distance(self):
    return self.max_distance

def get_size(self):
    return len(self.vertices)
```
The Recursive Function

```python
def tsp_solver_helper(tsp, path, distance, current_city, final_city):
    path = path + [current_city]
    if len(path) == tsp.get_size():
        return path + [final_city], distance + current_city.get_distance(final_city)
    best_distance = tsp.get_max_distance()
    best_route = None
    for city in city_gen(current_city, path):
        current_distance = distance + current_city.get_distance(city)
        temp_route, temp_distance = \
            tsp_solver_helper(tsp, path, current_distance, \
                city, final_city)
        if best_distance >= temp_distance:
            best_distance = temp_distance
            best_route = temp_route
    return best_route, best_distance
```
def solve_tsp(tsp, city):
    best_route, best_distance = tsp_solver_helper(tsp, [], 0, city, city)
    print_route(best_route)
    print(best_distance)

def print_route(route):
    for city in route:
        print(city.get_name(), end=" ")
    print()
Putting it All Together

city1 = GraphNode(1)
tsp = CompleteGraph(city1)
city2 = GraphNode(2)
tsp.add_vertex(city2, city1, 10)
city3 = GraphNode(3)
tsp.add_vertex(city3, city1, 15)
tsp.add_edge(city3, city2, 35)
city4 = GraphNode(4)
tsp.add_vertex(city4, city1, 20)
tsp.add_edge(city4, city2, 25)
tsp.add_edge(city4, city3, 30)

solve_tsp(tsp, city1)
There are $N!$ permutations of the order to visit the cities.

For each permutation, the recursive function is called $O(N)$ times.

Other functions complexity is negligible.
List Complexity Reminder

<table>
<thead>
<tr>
<th>Operation</th>
<th>Average Case</th>
<th>Amortized Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Append</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Insert</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Get Item</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Set Item</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Delete Item</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Iteration</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Get Slice</td>
<td>O(k)</td>
<td>O(k)</td>
</tr>
<tr>
<td>Del Slice</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Set Slice</td>
<td>O(k+n)</td>
<td>O(k+n)</td>
</tr>
<tr>
<td>Extend</td>
<td>O(k)</td>
<td>O(k)</td>
</tr>
<tr>
<td>Sort</td>
<td>O(n log n)</td>
<td>O(n log n)</td>
</tr>
<tr>
<td>Multiply</td>
<td>O(nk)</td>
<td>O(nk)</td>
</tr>
<tr>
<td>x in s</td>
<td>O(n)</td>
<td></td>
</tr>
<tr>
<td>min(s), max(s)</td>
<td>O(n)</td>
<td></td>
</tr>
<tr>
<td>Get Length</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
## Set Complexity Reminder

<table>
<thead>
<tr>
<th>Operation</th>
<th>Average case</th>
<th>Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>x in s</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Union $s\cup t$</td>
<td>$O(</td>
<td>s</td>
</tr>
<tr>
<td>Intersection $s\cap t$</td>
<td>$O(\min(</td>
<td>s</td>
</tr>
<tr>
<td>Multiple intersection $s_1 &amp; s_2 &amp; \ldots &amp; s_n$</td>
<td>$(n-1) \times O(L)$</td>
<td>$L = \max(</td>
</tr>
<tr>
<td>Difference $s-t$</td>
<td>$O(</td>
<td>s</td>
</tr>
<tr>
<td>s.difference_update(t)</td>
<td>$O(</td>
<td>t</td>
</tr>
<tr>
<td>Symmetric Difference $s^t$</td>
<td>$O(</td>
<td>s</td>
</tr>
<tr>
<td>s.symmetric_difference_update(t)</td>
<td>$O(</td>
<td>t</td>
</tr>
</tbody>
</table>
## Dict Complexity Reminder

<table>
<thead>
<tr>
<th>Operation</th>
<th>Average Case</th>
<th>Amortized Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Get Item</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Set Item</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Delete Item</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Iteration</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>
Recursive Function Complexity

def tsp_solver_helper(tsp, path, distance, current_city, final_city):
    path = path + [current_city]
    if len(path) == tsp.get_size():
        return path + [final_city], distance
    best_distance = tsp.get_max_distance()
    best_route = None
    for city in current_city.get_neighbors():
        if city not in path:
            current_distance = distance + current_city.get_distance(city)
            temp_route, temp_distance = \
            tsp_solver_helper(tsp, path, current_distance, city, final_city)
            if best_distance >= temp_distance:
                best_distance = temp_distance
                best_route = temp_route
    return best_route, best_distance
So the overall complexity is 
$O(N!*N*N^2)=O(N^3*N!)$

The problem is with the factorial factor

Best (optimal) algorithm avoids paths it can tell in advance are irrelevant. See https://en.wikipedia.org/wiki/Branch_and_bound

Still, can only solve for about 50 cities
def city_gen(current_city, path):
    for city in current_city.get_neighbors():
        if city not in path:
            yield city

def tsp_solver_helper(tsp, path, distance, current_city, final_city):
    path = path + [current_city]
    if len(path) == tsp.get_size():
        return path + [final_city], distance
    best_distance = tsp.get_max_distance()
    best_route = None
    for city in city_gen(current_city, path):
        current_distance = distance + current_city.get_distance(city)
        temp_route, temp_distance = \
        tsp_solver_helper(tsp, path, current_distance, \
                         city, final_city)
        if best_distance >= temp_distance:
            best_distance = temp_distance
            best_route = temp_route
    return best route, best distance
Questions From Previous Exams
Question From Moed A 2015

שאלה 2 (33 נקודות)

נתונה המוחלקת המميزة בין ברשים麦克ושת (סופית).

```python
class Node:
    def __init__(self, data, next_node=None):
        self.data = data
        self.next = next_node
```

נתונים שני אובייקטים מטיפוס Node, שנסמכים על ברשים י-ו, שهما ראשית של שתי ברשים麦克ושת.

המדרון: נ<count>ם כי בשתי ברשים麦克ושת המופיעים ב-ל חלוק שתי איברים משותפים בין ברשים.

לדומא, הראשה麦克ושת בברש麦克ושת שניל איברים משותפים דמויי-ו-ליא. נלך לבר משותף麦克ושת, המאיבר麦克ושת כל האיברים麦克ושת RETURNS麦克ושת הפ.getRequestDispatcher麦克ושת הפ分かりי בבר משותף.

def node_x is node_y म-ו-M, node_y מ-ו-M, node_x מ-ו-M, node_y מ-ו-M משותפים, אד-ו-ל麦克ושת הפ分かりי בבר משותף.
Question From Moed A 2015

# 2.1

```python
def len_list(head):
    length = 0
    while head is not None:
        length += 1
        head = head.next

    return length
```
Question From Moed A 2015

```python
# 2.2

def node_i(head, idx):
    if idx < 0:  # If the index is negative then idx = len+idx
        idx = len_list(head) + idx
        if idx < 0:
            return None

    while idx > 0 and head is not None:
        head = head.next
        idx -= 1

    return head
```
def is_tail(head_x, head_y):

# Your code here
# 2.3

def is_tail(head_x, head_y):
    if not head_x or not head_y:
        return False

    # Run until we reached the end of x or found a shared Node
    # We note that it's enough that we found one joint node to declare tail
    while head_x is not None and head_x != head_y:
        head_x = head_x.next

    if not head_x:
        # We reached the end of x
        return False

    return True
def find_joint_node(head_x, head_y):

...
```python
# 2.4

def are_joint(head_x, head_y):
    if not head_x or not head_y:
        return None

    len_x = len_list(head_x)
    len_y = len_list(head_y)

    cur_x = node_i(head_x, max(0, len_x - len_y))
    cur_y = node_i(head_y, max(0, len_y - len_x))

    # Now len_list(cur_x) == len_list(cur_y)

    # Run until we reached the end of the lists
    # or we found a joint Node.
    # Again it's enough to find one shared Node to declare joint lists
    while cur_x and cur_x != cur_y:
        cur_x = cur_x.next
        cur_y = cur_y.next

    if not cur_x:
        return None

    return cur_x
```
class Node(object):
    def __init__(self, data, next=None):
        self.data = data
        self.next = next

Not a mistake the variable is used as a keyword in Hebrew.

None is a keyword in Hebrew.

Changing the variable name to a different Hebrew keyword (in Hebrew, a method with a different Hebrew name) is suggested.

This is also not a mistake.
def has_cycle(head):
    past = []
    cur = head
    while cur is not None:
        if cur in past:
            return True
        past.append(cur)
        cur = cur.next
    return False
סעיף השאלת הבימה什么东西 מתוכך לפתרוןalemון beinotel שלูילירי של ביעי 3 (כלומר בסיסויות (ח), זמקום קבוש.
ובכל שלב יתק לועשה שימשו ברעיונות מספרים Коודקים זכר לא בהכרח עליל לקורא ישימם ל久しぶり שנותב בהפעלי Коודקים.
ב. (5 Коודקים)
המקבלת כראהemento שימשמקושטרת (нтונה כמצביע לאיבר ההרשוק node_i וממש את הפוקצית המיאודקסים מהיאיבר ברישה) ויאנדים וממדיו מדעי לאיבר h ברישה. שימה לב כי ראש התשומת מודר להיווי האיבר עם אינדקס אפס ברישה (0=i). אם התשומת ריקה או שיאנדקסו i מחקו לטווח האינדקסים ברישה, לעיכםelahזיא:

def node_i(head,i):

Question From Moed B 2015

```python
#part b:
def node_i(head, i):
    if i < 0:
        return None
    while head and i > 0:
        i -= 1
        head = head.next
    return head
```
Question From Moed B 2015

ג. (9 נקודות)

בהתחשב בתפקיד קבוצת השם שמיימה \( cycle\_length \) המגדירה את האינדקס של המונח החציוני של האינדקס של המונח השני של האיבר בראש הרשימה בטור הרשימה. נתון ליניארי כי ברשימה המתקדמת לא קיימים איברים בתור מהשניים. התוכנית \( cycle\_length(\text{head}) > 0 \) ו\( \text{node}_i(\text{head}, cycle\_length(\text{head})) \) הוא \text{head}. באים \( cycle\_length \). True. התחזיות ההפוקציה \( cycle\_length(\text{head}) \):

```python
def cycle_length(head):
```
Question From Moed B 2015

```python
#part c: (assumes list is circular, non-empty, and returns to head)
def cycle_length(head):
    result = 1
    cur = head.next
    while cur is not head:
        cur = cur.next
        result += 1
    return result
```
דר (10 נקודות)

שימו לב, בסעיף זה יש למעאים פתרוןشرקבדמנזרחלארייבמקומכםkB

המכבולת carreraומנותreshimamkoshretומדהירהאתיועכןתקוןביוער

םאתההפניהיהעיבריה-ברשימהיהלאнныеה1+2ברשיםיה.ברתם,ערךיהбинויי

node_i(head,i) is node_i(head,2*i+1)

haos isha.ינחילולניהיכישברשמתמקומכםכד.

def find_loop(head):

...
#part d: assumes there is an index i in the list such that
#the i'\th item and the (2i+1)'th item are the same

def find_loop(head):
    i = 0
    first = head
    second = head.next
    while first is not second:
        i += 1
        first= first.next
        second = second.next.next
    return i
ה. (5 נקודות)

הנה מצודת את הפונקציה \( has_cycle \) המסיעת את, "כRestController לדמי ולני ולייר, ובמקומם קבע.

 Ramadan: ישוב של הלהבה של סעיף. איך היא מתכיות כשאני מתעמל?
#part e: This part relies strongly on the solution to the previous part.
# if the condition of part d does occur, then we have a loop, otherwise we do not.
# we just need to change return values to True/False instead of the index i, and we
# must be careful not to "fall off the edge" of the list if there is no loop.
def has_cycle(head):
    if not head or not head.next:
        return False
    first = head
    second = head.next
    while first is not second:
        first = first.next
        if not second.next or not second.next.next:
            return False
        second = second.next.next
    return True