Lecture 3: Recursion and Divide and Conquer

In Python a function can call itself. If a function calls itself, it is called a recursive function. Recursion may also correspond to a function $A$ calling a function $B$, which in turn calls function $A$. We’ll focus on the simple case of recursion here, namely a function $f$ calling $f$ again.

Merge Sort

Let’s look at the recursive Divide-and-Conquer paradigm. We will first describe how it is used in Merge Sort, a popular sorting algorithm.

Suppose we have a list as shown here that needs to be sorted in ascending order:

```
a b c d
```

We divide it into two equal-sized sublists:

```
a b
```
```
c d
```

Then, we sort the sublists recursively. At the base case of the recursion, if we see a list of size 2, we simply compare the two elements in the list and swap if necessary. Let’s say that $a < b$, and $c > d$. Since we want to sort in ascending order, after the two recursive calls return we end up getting:

```
a b
d c
```

Now we come back to the first (or topmost) call to sort and our task is to merge the two sorted sublists into one sorted list. We do this using a merge algorithm, which simply compares the first two elements of the two sublists, repeatedly. If $a < d$, then it first puts $a$ into the merged (output) list and effectively removes it from the sublist. It leaves $d$ where it was. It then compares $b$ with $d$. Let’s say that $d$ is smaller. Merge puts $d$ into the output list next to $a$. Next, $b$ and $c$ are compared. If $c$ is smaller, $c$ is placed next into the output list, and finally $b$ is placed. The output will be:

```
a d c b
```
Note that if the number of elements is odd, the subarrays will not be equal in size but differ in size by 1.

Below is code for Merge Sort that can be found in `mergesort.py`.

```python
1. def mergeSort(L):
2.     if len(L) == 2:
3.         if L[0] <= L[1]:
4.             return [L[0], L[1]]
5.         else:
6.             return [L[1], L[0]]
7.     else:
8.         middle = len(L)//2
9.         left = mergeSort(L[:middle])
10.        right = mergeSort(L[middle:]):
11.        return merge(left, right)
```

Lines 2-6 correspond to the base case where we have a list of two elements and we place them in the right order. If the list is longer than two in length, we split the list in two (Line 8), and make two recursive calls, one on each sublist (Lines 9-10). We use list slicing; `L[:middle]` returns the part of the list `L` corresponding to `L[0]` through `L[middle-1]`, and `L[middle:]` returns the part of the list corresponding to `L[middle]` to `L[len(L)-1]`, so no elements are dropped. Finally, on Line 11, we call `merge` on the two sorted sublists and return the result.

All that remains is the code for `merge`.

```python
1. def merge(left, right):
2.     result = []
3.     i, j = 0, 0
4.     while i < len(left) and j < len(right):
5.         if left[i] < right[j]:
6.             result.append(left[i])
7.             i += 1
8.         else:
9.             result.append(right[j])
10.        j += 1
11.     while i < len(left):
12.         result.append(left[i])
13.        i += 1
14.     while j < len(right):
15.         result.append(right[j])
16.        j += 1
17.     return result
```

Initially, `merge` creates a new empty list `result` (Line 2). There are three `while` loops in `merge`. The first one is the most interesting, which corresponds to the general case when the two sublists are both nonempty. In this case, we compare the current first elements of
each of the sublists (represented by counters \( i \) and \( j \), pick the smaller one, and increment the counter for the sublist whose element we selected to place in result. The first \texttt{while} loop terminates when either of the sublists becomes empty.

When one of the sublists is empty, we simply append the remaining elements of the nonempty sublist to the result. The second and third \texttt{while} loops correspond to the left sublist being nonempty and the right one being nonempty.

\textbf{Merge Sort Execution and Analysis}

Suppose we have this input list for Merge Sort:

\[
\text{inp} = [23, 3, 45, 7, 6, 11, 14, 12]
\]

How does the execution proceed? The list is split into two:

\[
[23, 3, 45, 7] \quad [6, 11, 14, 12]
\]

and the left list is sorted first, and the first step is to split it in two:

\[
[23, 3] \quad [45, 7] \quad [6, 11, 14, 12]
\]

Each of these 2-element lists is sorted in ascending order:

\[
[3, 23] \quad [7, 45] \quad [6, 11, 14, 12]
\]

The sorted lists of two elements each are merged into a sorted result:

\[
[3, 7, 23, 45] \quad [6, 11, 14, 12]
\]

Next, it looks at the right list and splits it in two:

\[
[3, 7, 23, 45] \quad [6, 11] \quad [14, 12]
\]

Each of the right sublists is sorted:

\[
[3, 7, 23, 45] \quad [6, 11] \quad [12, 14]
\]

The sorted sublists of two elements each are merged into a sorted result:

\[
[3, 7, 23, 45] \quad [6, 11, 12, 14]
\]

And finally, the two sorted sublists of 4 elements each are merged:

\[
[3, 6, 7, 11, 12, 14, 23, 45]
\]
Merge Sort is a more efficient sorting algorithm than the Selection Sort algorithm we described in the celebrity selfie tutorial (Tutorial 1). Selection Sort had a doubly nested loop and therefore required in the worst-case \( n^2 \) comparisons and exchanges for a list of length \( n \). In Merge Sort, we only perform \( n \) operations in each merge step of a list of length \( n \). The top-level merge will require \( n \) operations, and the next level will run merge on two lists of length \( n/2 \), and also require \( n \) operations. The next level after that will require \( n/4 \) operations on each of four lists, for a total of \( n \) operations. The number of levels is \( \log_2 n \), and hence Merge Sort requires \( n \log_2 n \) operations.

As you can see during the merge step we need to create a new list \( \text{result} \) (Line 2) and return that. The amount of intermediate storage required for Merge Sort therefore grows with the length of the list to be sorted. This was not the case for Selection Sort, which we saw in the celebrity selfie tutorial code.

**In Place Merging**

It is possible to perform an *in place* merge sort, where only a fixed amount of intermediate storage is required, and a copy of the list need not be made. Such a merge routine is given in `mergesort-inplace.py`. *The algorithm for merge is itself recursive and is described below.*

We’ll first describe rotations as they are used heavily in the in place merging process. For in-place merging, rotations are used to swap the elements in two *adjacent* sublists, which are *equal in length*. As you will soon see, this technique is used to move a large number of elements at once. See the example below where we have performed a swap (via a rotation as you will see).

![Example of In Place Merging](image)

The above is the basic operation of in-place merging. To rotate elements in a sublist means to shift the elements some number of places to right and elements at the end wrap

---

around to the beginning (and vice versa). So if you have the list [1,2,3,4,5], shifting it one place to the right will give you [5,1,2,3,4].

The fact that the two ranges being swapped are adjacent is important in calling this operation a rotation. In the following graphic, the elements are shifted one place to the right three times, achieving the same result as before.

Now we can explain how to use rotations for in-place merging. Let us define two sorted sublists, A and B that we want to merge in-place.

\[
A = \begin{array}{ccccccc}
10 & 15 & 19 & 23 & 26 & 32 \\
\end{array}
\]

\[
B = \begin{array}{ccccccc}
4 & 9 & 16 & 30 & 39 & 47 \\
\end{array}
\]

The trick is to swap the largest elements in A for the smallest elements in B. A linear search can find the range of elements to rotate. Start from the middle and extend outwards, until you find an element in A which is less than an element in B. Rotate the elements between these bounds.
That’s the basic operation. However, something special happened. Of the two ranges, the smallest elements are in A and the largest elements are in B, albeit not in order. This means you can continue to merge A and B independently, so all you’re left with is two smaller merges. This also means that in-place merging is a recursive function in itself.

By **recursively** applying rotations in this manner, eventually you’ll merge the two ranges. Let’s apply this process once more and see what happens:

Just like that, the right-hand side is already sorted and the left-hand side is almost sorted. Two more rotations on the left-hand side and the merge procedure would be complete.

While this is an elegant algorithm and worth understanding, it turns out in place merging is not particularly useful. This is because the number of swaps required can grow in the worst case as $n^2$ where n is the number of elements in the list. In place merge sort is therefore less efficient than allocating memory and using n comparisons for the merge ☺
Courtyard Tiling

Consider the following tiling problem. We have a courtyard with \(2^n \times 2^n\) squares and we need to tile the courtyard using L-shaped tiles or trominoes. Each trominoe consists of three square tiles attached to form an L shape as shown below.

\[
\begin{array}{c}
\begin{array}{c}
2^n \\
\end{array}
\end{array}
\]

Can this be done without spilling over the boundaries, breaking a tromino or having overlapping trominoes? The answer is no, simply because \(2^n \times 2^n = 2^{2n}\) is not divisible by 3, only by 2. However, if there is one square that can be left untiled, then \(2^{2n} - 1\) is divisible by 3. Can you show this?

We, therefore, have hope of properly tiling a \(2^n \times 2^n\) courtyard with one square that we can leave untiled because, for example, there is a statue of your favorite President on it. We’ll call this square that can be left untiled the missing square.

**Is there an algorithm that tiles any \(2^n \times 2^n\) courtyard with one missing square in an arbitrary location?** As an example, below is a \(2^3 \times 2^3\) where the missing square is marked \(\Delta\). Does the location of the missing square matter?

\[
\begin{array}{c}
\begin{array}{c}
\Delta \\
\end{array}
\end{array}
\]

---

\(^2\) Since this is not a class on mathematical puzzles we’ll tell you! \(2^{2n} - 1\) can be written as \((2^n - 1)(2^n + 1)\). \(2^n\) is clearly not divisible by 3. And that means that either \(2^n - 1\) is divisible by 3 or \(2^n + 1\) is divisible by 3.
Base Case of a $2 \times 2$ courtyard

Let's start with $n = 1$, i.e., a $2 \times 2$ courtyard with one missing square. The missing square can be in different locations as shown below with the $\Delta$'s in different locations. In all four cases as shown below, we can tile the remaining 3 squares with an L-shaped tile oriented appropriately.

![Diagram of four cases]

This is an important step because we now have a base case for a recursive Divide-and-Conquer algorithm that we can apply. But how to divide the $2^n \times 2^n$ courtyard with one missing square, so we get subproblems that are smaller in size that are the same problem except smaller? If we divide the $2^n \times 2^n$ courtyard into four $2^{n-1} \times 2^{n-1}$ courtyards as shown below, we get one $2^{n-1} \times 2^{n-1}$ courtyard with a missing square – the one on the top right – but the other three are full $2^{n-1} \times 2^{n-1}$ courtyards.

Recursive Step

In the above example, we recognize that the statue $\Delta$ is in the top right quadrant. Therefore we place a tromino strategically on the three “full” quadrants to create a set of four quadrants each with one missing square that need to be tiled.

![Diagram of recursive step]

The top right quadrant is unchanged, but the other three have exactly one square tiled, and so the work that remains to be done is to tile four $2^{n-1} \times 2^{n-1}$ courtyards with one.
missing square each. Sound familiar? Note that if the missing square where in the top left quadrant we would rotate the tromino counterclockwise by 90 degrees and obtain four smaller courtyards as before. Work out the rotations for the other two cases.

We do this recursively till we obtain $2 \times 2$ courtyards with one missing square. Each of these is trivially tiled with one tromino as we showed earlier. We’re done.

Not quite. You have a large $2^n \times 2^n$ courtyard with a statue to tile and tiling with L-shaped tiles is a complicated job. This means you have to be very specific about each tile that needs to laid down to the flooring contractor. We need to write a program that will generate a “map” of all of the tiles for this courtyard. Given that this is a book about programming, we hope you didn’t think that we were going to stop with a vaguely specified algorithm!

We are going to number quadrants in a particular way as shown below in the code that we write. Our courtyard is going to be represented as $yard[r][c]$, where $r$ is the row number and $c$ is the column number. The row numbers increase from top to bottom, and column numbers increase from left to right as shown below.
The function `recursiveTile` takes as arguments a two-dimensional grid `yard`, the row and column coordinates of the origin and the dimension of the current yard (or quadrant) being tiled, and the location of the missing square (`rMiss, cMiss`) in relation to the origin. It also takes a helper variable `nextPiece` as a last argument which is used to number tiles so we can print a map that the contractor will find easy to read. Assume for now that the origin is `(0, 0).

The quadrant with the missing square is identified based on the location of the missing square on Line 2 shown below:

\[
\text{quadMiss} = 2*(rMiss \geq size/2) + (cMiss \geq size/2)
\]

Rows are numbered from 0 at the top to size - 1 at the bottom. Columns are numbered 0 at the left to size - 1 on the right. As an example, if `rMiss = 0`, and `cMiss = size - 1`, we have the missing square in the top right quadrant, and `quadMiss` is computed as
2 * 0 + 1 = 1. For large rMiss and cMiss equal or greater than size/2 we will get the lower right quadrant numbered 3.

The base case is written first in recursiveTile from Lines 3-9 and shows how nextPiece is used to mark the squares on the board with the number of the tile that was used to cover the squares. The base case is for a 2 × 2 courtyard with one missing square in square quadMiss. We know we can tile the courtyard regardless of what quadMiss is, and we number the tile using nextPiece and fill in the courtyard yard. Since we do not want to tile the square quadMiss (it has either been tiled already or will have a statue on it), we remove it from the list of tuples piecePos using the pop function on the list. The pop function takes the index of the element in the list as an argu-

The overall recursion works as follows. Based on quadMiss, four recursive calls are made (Lines 10-18); quadMiss is the location of the missing square in the yard, and the quadrant corresponding to quadMiss will have the missing square. However, in the three other quadrants, we will also have a missing square at one of the corners since we will eventually place a tile at those corners. This center tile is placed after the recursive calls return (Lines 19-23).

There is some computation to determine the arguments for the recursive calls. The size of the courtyard is size/2 and we pass nextPiece into the recursive calls. We also need to correctly compute the origin coordinates for each of the four quadrants. The shifts required in the coordinates are computed in Lines 11-12. Given our numbering of the quadrants, when we make the recursive calls, Quadrants 0 and 1 and will have the same origin row coordinates as in the parent procedure, and Quadrants 2 and 3 will have their origin row coordinates shifted by size/2 in relation to the parent procedure (Line 11). Quadrants 0 and 2 will have the same origin column coordinates as the parent procedure and Quadrants 1 and 3 will have origin column coordinates shifted by size/2 (Line 12).

For the recursive call corresponding to the quadrant quadMiss that has the square that was missing in the parent call, we simply compute the new rMiss and cMiss in relation to the shifted origin coordinates (Line 14).

The computation of the rMiss and cMiss arguments is different for the other three recursive calls because the missing square is going to be at one of the corners and is unrelated to the values of rMiss and cMiss in the parent call. The computation is given in Lines 16-17. For Quadrant 0, the bottom right corner will be the missing square and its coordinates in relation to the origin are rMiss equaling size//2 – 1, and cMiss equaling size//2 – 1. Similarly, for the other quadrants.

Finally, in Lines 19-23, we place the center tile in the yard, again based on quadMiss, which points to the square on which the tile should not be placed. We could just have
easily placed the center tile prior to making the recursive calls. The only difference this would make would be in the numbering of the tiles, not where tiles are placed. Lines 19 and 19a show the creation of a list centerPos in Python list comprehension style – more on that below. This list initially contains the four middle squares in the courtyard being processed in this call, which are each corner squares of the four quadrants of the courtyard. Line 20 removes the corner square that belongs to the quadrant containing quadMiss from centerPos.

Let’s now look at how to invoke recursiveTile.

1. \text{EMPTYPIECE} = -1
2. \textbf{def} tileMissingYard(n, rMiss, cMiss):
3. \hspace{1em} \text{yard} = \left[ [\text{EMPTYPIECE} \ 	ext{for} \ i \ \text{in} \ \text{range}(2**n)]
3a. \hspace{2em} \text{for} \ j \ \text{in} \ \text{range}(2**n)]
4. \hspace{1em} \text{recursiveTile(yard, } 2**n, 0, 0, \text{rMiss, cMiss, 0)}
5. \hspace{1em} \text{return yard}

We will use -1 to signify an empty square in the yard (Line 1). The function tileMissingYard is simply a wrapper for the function recursiveTile, which needs the arguments corresponding to the origin coordinates and nextPiece initialized to 0. On Lines 3 and 3a, we create a new two-dimensional list corresponding to a yard with each dimension equaling $2^n$ and initialize it in Python list comprehension style, which is more compact than the standard nested for loops that we would need to fill in a two-dimensional list. We emphasize that we allocate no memory for courtyards within the recursiveTile procedure – each recursive call fills in disjoint parts of the variable yard that is passed into the procedure.

The returned nextPiece value is ignored for the top-level call to recursiveTile but used in subsequent recursive calls.

The code can be found in tiling.py.

**List Comprehension Basics**

List comprehensions can be used to create lists in a natural way. Suppose we want to produce the lists $S$ and $M$ defined mathematically below:

\[
S = \{x^3 : x \in \{0 \ldots 9\}\}
\]
\[
O = \{x | x \in S \text{ and } x \text{ odd}\}
\]

Here’s how we can produce the lists using list comprehensions:

\[
S = [x**3 \ \text{for} \ x \ \text{in} \ \text{range}(10)]
\]
\[
O = [x \ \text{for} \ x \ \text{in} \ S \ \text{if} \ x \ % \ 2 \ == \ 1]
\]
The first expression in the list definition corresponds to an element of the list and remaining part of the definition generates list elements according to specified properties. We are including all cubes of numbers between 1 and 9 in S. We are including all odd numbers in S in the list O.

Here’s a more interesting example where we compute the list of prime numbers less than 50. First, we build a list of composite numbers, using a single list comprehension, and then we use another list comprehension to get the “inverse” of the list, which are the prime numbers.

```python
composites = [j for i in range(2, 8) for j in range(i*2, 50, i)]
primes = [x for x in range(2, 50) if not x in composites]
```

The two loops in the definition of composites find all the multiples of numbers between 2 and 7 that are less than 50. The number 7 was chosen because $7^2 = 49$, the highest number less than 50. Some composite numbers may be repeated in the composites list. The definition for primes simply walks through numbers between 2 and 49 and includes numbers that are not present in the composites list.

List comprehensions can produce very compact code, which is sometimes incomprehensible 😳 Use them in moderation!

### Pretty Printing

One reason for this coding exercise was to produce a contractor-friendly map and here is the printing routine that does exactly that.

```python
1. def printYard(yard):
2.     for i in range(len(yard)):
3.         row = ''
4.         for j in range(len(yard[0])):
5.             if yard[i][j] != EMPTYPIECE:
6.                 row += chr((yard[i][j] % 26) + ord('A'))
7.             else:
8.                 row += '
9.     print (row)
```

The printing procedure prints the two-dimensional yard on one line per row. It creates a row of characters corresponding to the tiles on that row and prints the row. There is one missing square in the courtyard that is left empty by printing a space (Line 8).

We could print the numbers, but we chose to replace them with letters A through Z. The function `chr` takes a number and produces a letter associated with that number in ASCII format, and the function `ord` is the inverse of `chr` in that it takes a letter and produces its ASCII number. The tile with number 0 is assigned the letter A on Line 5, and the tile with number 1 is assigned the letter B, and so on till number 26 gets Z. If the
court yard is \(2^5 \times 2^5\) or larger, we will have more than 26 L-shaped tiles and the some tiles will get the same letter. (Of course, we could have printed these tiles’ unique numbers. One issue with printing numbers is that we would have to deal with single-digit and two-digit numbers during printing.)

Let’s run:

```python
printYard(tileMissingYard(3, 4, 6))
```

to get:

```
AABBFFGG
AEEBFJJG
CEDHHJI
CCDUUHII
KKLUPP Q
KOLLPTQQ
MOONRTTS
MMNNRRSS
```

This is descriptive enough for any contractor to tile the courtyard properly! You can see in what order the tiles are laid by `recursiveTile`. The recursive calls follow the order of the quadrants 0, 1, 2 and 3 (Line 12 of `recursiveTile`). The first tile that is laid is tile A, which corresponds to the top left quadrant. The center tile U is laid last since we chose to lay down the center tile after the recursive calls returned. If we lay the center tile before making the recursive calls, the center tile would have been A.

Let’s do a runtime analysis of `recursiveTile`. It runs fairly fast on large yards. The key observation is that the yard size, i.e., the length and width of the courtyard in each recursive call is half of the original. This means that if we start with a \(2^n \times 2^n\) courtyard, in \(n-1\) steps we will get to the base case of a \(2 \times 2\) courtyard. Of course, at each step, we are making four recursive calls, giving us \(4^{n-1}\) calls, each of which tiles a \(2 \times 2\) courtyard. The number of squares processed is exactly \(2^n \times 2^n\), which is the number of squares in the initial courtyard. Of course, one of the squares is not tiled.