Algorithm design practice

Agenda

- Algorithm design strategies
  - Induction
  - Recursion and Divide & Conquer
  - Dynamic programming
- Algorithm design exercises

Induction

Induction

Induction is intuitive to us.

- First, showing a solution is feasible for a base case (basis step)
- Then, showing it carries over from one object to the next (inductive step)

Fibonacci sequence

The Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

- Each number (except the first two) is the sum of the two numbers that precede it.

Finding the n-th Fibonacci number.

So, we know that

- 3rd Fibonacci number is 0+1 = 1
- 4th Fibonacci number is 1+1 = 2
- 5th ...
- ……

Actually, from the definition, we know that f(n) = f(n-1) + f(n-2).

Bottom-up: Fibonacci number

```
func fib_bottom_up(n) #n>2
    var f as an array of size n
    f[0] = 0
    f[1] = 1
    for i from 2 to n-1:
        f[i] = f[i-1] + f[i-2]
    return f[n]
end func
```

Bottom-up: generating Fibonacci number inductively

```
def fib(n):
    """
    n > 0
    This function can return the
    fib sequence.
    """
    if n == 1:
        return 0
    f = [0] * n
    f[1] = 1
    for i in range(2, n):
        f[i] = f[i-1] + f[i-2]
    return f[n-1]
```

```
def fib(n):
    """
    n > 0
    return the n-th fib number
    """
    a = 0
    b = 1
    if n == 1:
        return a
    if n == 2:
        return b
    for i in range(2, n):
        c = b + a
        a = b
        b = c
    return c
```

Finding two closest but not identical numbers in a list

Given a list of numbers(n>2), you need to find the two (nonidentical) numbers that are closest to each other.

One intuitive solution:

- When n = 3, [a, b, c] → We calculate the absolute value of |a-b|, |a-c|, |b-c|, the closest numbers have a smallest absolute value.
- When n = 4, [a, b, c, d] → We calculate the absolute value of |a-b|, |a-c|, |a-d|, |b-c|, ..., the closest numbers have a smallest absolute value.

Our solution works for any list whose length is greater than 2.

Finding two closest but not identical numbers in a list

Following the intuitive solution, we can write the code like the right.

- Two nested loop: $O(n^2)$
- Stores the values in a dictionary.
  - It wastes memory space since we only need to find the first closest pair.
- Returns the first detected closest pair.

```
def find_closest(lst):
    d = {}
    for x in lst:
        for y in lst:
            if x == y:
                continue
            try:
                d[abs(x-y)].append((x, y))
            except KeyError:
                d[abs(x-y)]=[(x, y)]
    keys = sorted(d)
    return d[keys[0]][0]
```

A memory saving version

```
def find_first_closest(lst):
    dd = float("inf")
    for x in lst:
        for y in seq:
            if x == y:
                continue
            d = abs(x-y)
            if d < dd:
                xx, yy, dd = x, y, d
    return (xx, yy)
```

Finding two closest but not identical numbers in a list

```
def find_first_closest(lst):
    dd = float("inf")
    for x in lst:
        for y in seq:
            if x == y:
                continue
            d = abs(x-y)
            if d < dd:
                xx, yy, dd = x, y, d
    return (xx, yy)
```

- Our first solution is to calculate all absolute values between every items.
- Complexity is $O(n^2)$.

Actually, we can do it better.

- We sort the list first.
- Then, we only need to compare two neighboring items because the closest number must be neighbors in a sorted list.

Finding two closest but not identical numbers in a list

We reduce the original problem to the problem of sorting a sequence.

- The complexity is O(nlog(n) + n) ~ O(nlog(n))

```
def find_first_closest(lst):
    dd = float("inf")
    lst.sort()
    for i in range(len(lst)-1):
        x, y = lst[i], lst[i+1]
        if x == y:
            continue
        d = abs(x-y)
        if d < dd:
            xx, yy, dd = x, y, d
    return xx, yy
```

Recursion and Divide and Conquer

Recursion

In algorithm design, recursion is a technique which calls itself with “smaller (or simpler)” tasks.

- If a task can be solved by utilizing its solution to the smaller versions of the same tasks, then we can use recursion to solve it.

Generally, recursion can be applied in two circumstances,

- Recursion in induction
- Recursion in Divide and conquer

Recursion in induction

- Let F(n) be the problem of scale n.
- In induction, when we want to find the solution of F(n), we need to find the solution F(n-1), while to find F(n-1), we need to find F(n-2), and so on, until we find P(1).
- If the ways to find the solutions from F(n) to F(1) are the same, we can use recursion to solve the problem.

  This is what we called recursion in induction.

Recursion: Fibonacci number

- To find the n-th Fibonacci number f(n), we need to find f(n-1) and f(n-2).
- To find f(n-1), we need to find f(n-2), f(n-3).
- ...
- We stop until we reach f(2).

```
def fib_recursion(n):
    """
    Assum n is an integer n > 0
    """
    if n == 1:
        return 0
    if n == 2:
        return 1
    return fib_recursion(n-1) + fib_recursion(n-2)
```

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Recursion: Powerset

To find the powerset of set S = {1, 2, 3},

- We need to find the power set of {2, 3}.
- Once we get the powerset of {2, 3}, we generate the subsets that contains {1}.

To find the powerset of set S = {2, 3},

- We need to find the power set of the subset of {3}.
- Once we get the powerset of {3}, we generate the subsets that contains {2}.

..... We will stop until S is empty.

Recursion: Powerset

Following the discussion in the previous page, we program the solution using recursion.

Finding the powerset of the subset

Inserting the item

```
def powerset_recursive(lst):
    if len(lst) == 0:
        return [[]]
    pset = powerset_recursive(lst[1:])
    new_subset = deepcopy(pset)
    for subset in new_subset:
        subset.append(lst[0])
    pset.extend(new_subset)
    return pset
```

```
##recursion
def powerset_recursive(lst):
    if len(lst) == 0:
        return [[]]
    pset = powerset_recursive(lst[1:])
    new_subset = [subset + [lst[0]] for subset in pset]
    return pset + new_subset
```

Divide and conquer

Divide and conquer is based on decomposing the problem in a way that improves performance.

- The task of scale n is divided into k independent subtasks, each subtask can be solved in the same way of the origin task.
- Recursively solve the subtasks
- Combine the results, and thereby conquer the problem.

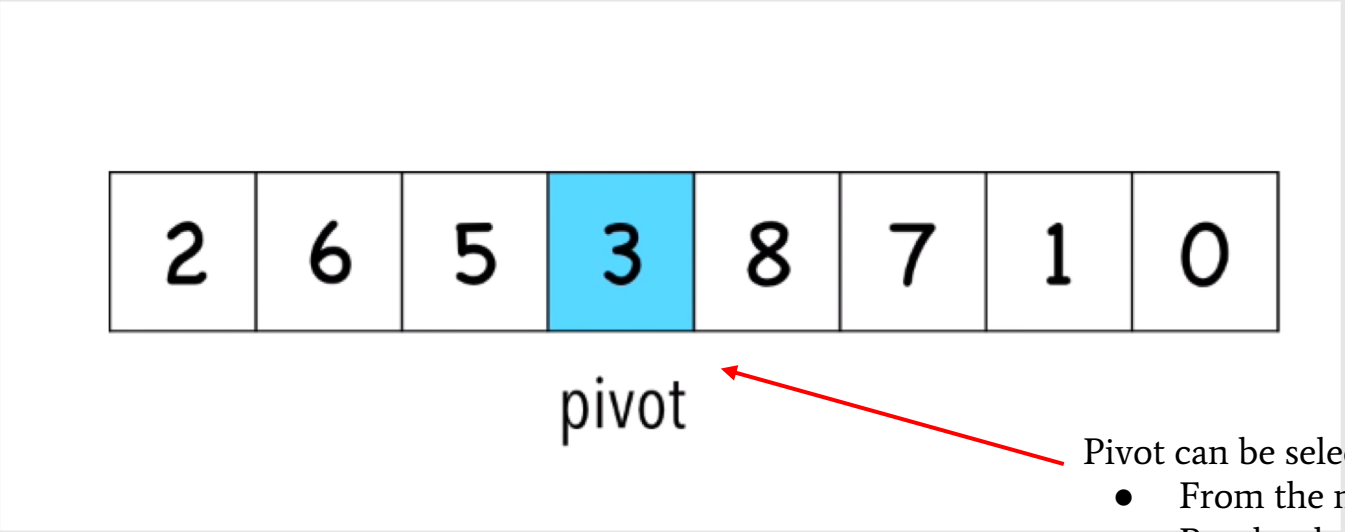
Example: Merge sort

Basic Idea: (divide and conquer)

1. Divide list into lists of single elements – which are by nature sorted
2. Merge adjacent lists to form sorted lists
3. Go to 2; continue with adjacent lists until one list is obtained

Quick sort

Step 1. Pick an element, called a pivot, from the array.



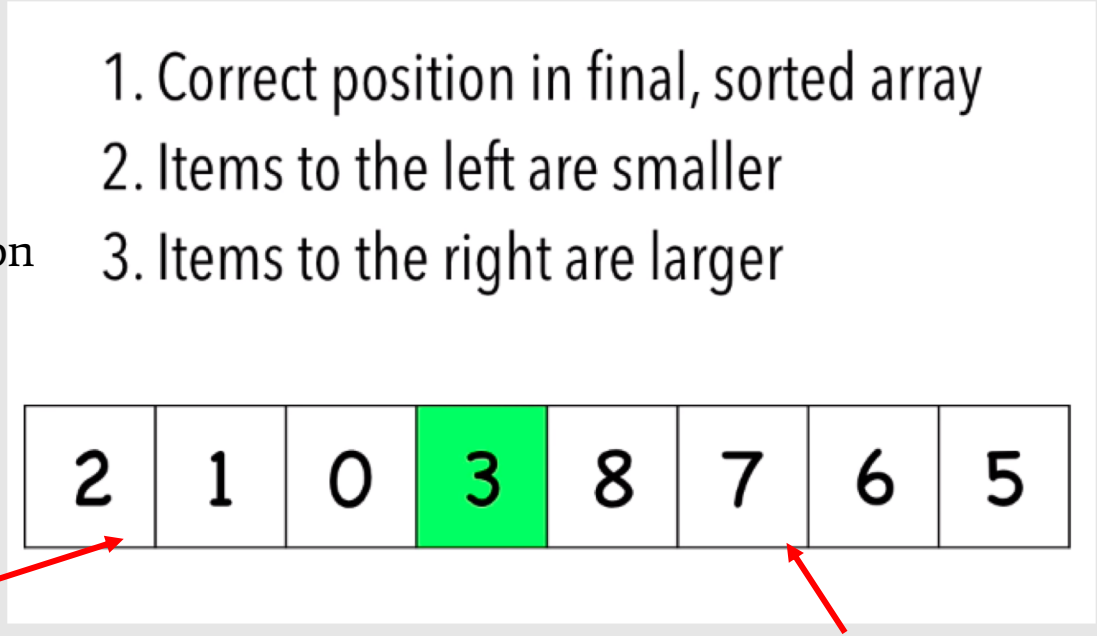
Pivot can be selected in many ways:

- From the most right/left
- Randomly choose one
- ...

Quicksort

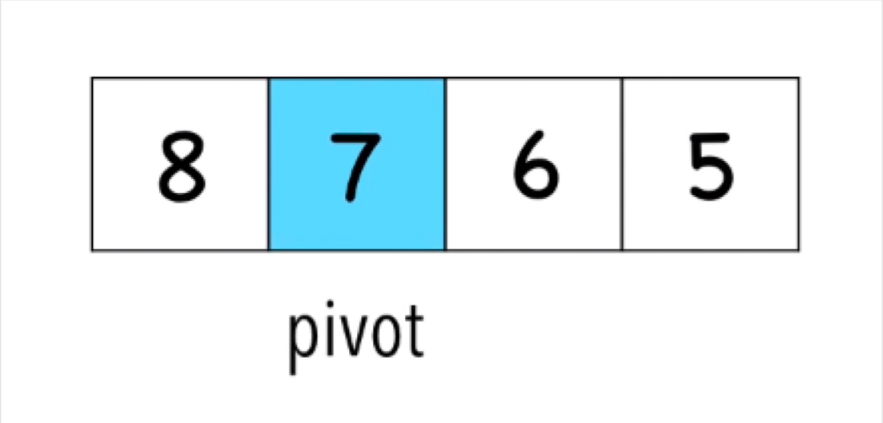
Step 2. Partitioning:

- reorder the array
- the pivot is in its final position after partitioning



Quicksort

Step 3. Recursively apply Step1 and Step 2 to the sub-array.



Quick sort

- Quicksort divides a large array into two small sub-array and a pivot.
  - the low elements
  - the pivot
  - the high elements
- It sorts the sub-arrays recursively.

Python code

```
def quicksort(seq):
    if len(seq) <= 1:
        return seq
    low, pivot, high = partition(seq)
    return quicksort(low) + [pivot] + quicksort(high)

def partition(seq):
    """complete the function"""
    pivot, seq = seq[0], seq[1:] #pick the first element as pivot
    low = []
    high = []

    # iterate through each element of the list and put it in either

    return low, pivot, high
```

Divide the problem into three sub-problem

Combine the results of each sub-problem

Input list: [30，24，5，58，18，36，12，42，39]

1.i=low, j =high, pivot=30

i								j
30	24	5	58	18	36	12	42	39

2. Move j to the left

such that R[j]<pivot

Swap R[i] and R[j]

i++

i						j		
30	24	5	58	18	36	12	42	39

	i					j		
12	24	5	58	18	36	30	42	39

3. Move i to the right

such that R[i]>pivot

Swap R[i] and R[j]

j--

			i			j		
12	24	5	58	18	36	30	42	39

			i		j			
12	24	5	30	18	36	58	42	39

4. Move j to the left

such that R[j]≤pivot

Swap R[i] and R[j]

i++

			i	j				
12	24	5	30	18	36	58	42	39

				i&j				
12	24	5	18	30	36	58	42	39

Dynamic programming

Fibonacci sequence

Computing the n-th Fibonacci number f(n) has optimal substructure:

- The f(n) is based on the solutions of f(n-1), f(n-2), ... , f(1).

Computing the n-th Fibonacci number f(n) has overlapping subproblems:

- Computing f(n-1) and f(n-2) both need to compute f(n-3)

So, we can solve it by using dynamic programming.

Fibonacci number: Recursion

```
def fib(n):
    """Assumes n is an int >= 0
       Returns Fibonacci of n"""
    if n == 0 or n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

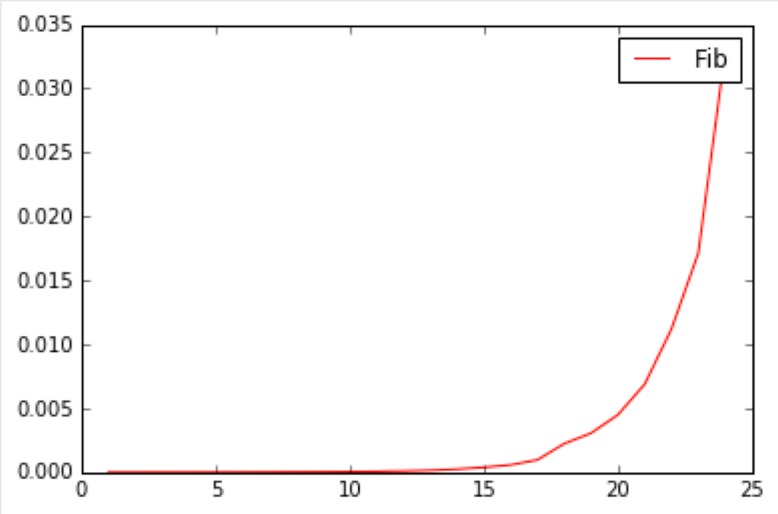
Complexity of fib_recursive(n) (Optional)

For the recursive version, the time complexity is $O(2^n)$.

Let T(n) be the time complexity of fib_recursive(n). The time complexity can be written as,

$T(n) = T(n-1) + T(n-2) + c$ # c is a constant
$\approx 2(T(n-1)) + c$ # approximation
$= 2(2T(n-2) + c) + c$
$= \ldots\ldots$
$= 2^kT(n-k) + (2^k-1)c$

When k = n, program stops, so the complexity upper bound is $O(2^n)$.



Complexity of fib_recursive(n)

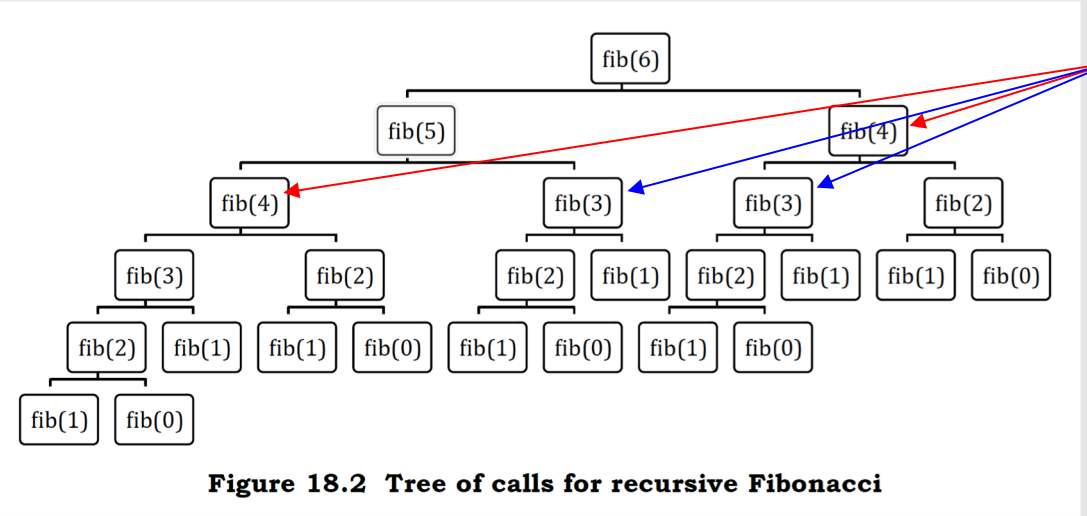


Figure 18.2 Tree of calls for recursive Fibonacci

Lots of redundant computation!

fast fib()

- Since there are overlaps between the subproblems, you can remember the past as you explore.

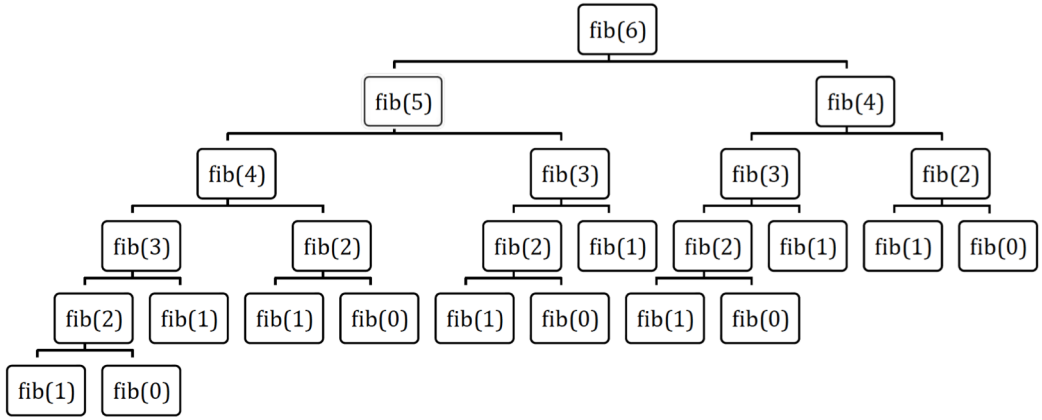


Figure 18.2 Tree of calls for recursive Fibonacci

- Finding fib(6) can be done by finding fib(5) and fib(4).
- The two subproblem has some overlaps, i.e., the result of fib(5) depends on the result of fib(4).
- We can save (remember) the result of fib(4)

fast_fib()

- Remember the past as you explore

```
def fast_fib(n, memo = {}):
    """
    Assume n is an int > 0, memo used only by recursive calls
    returns Fibonacci of n.
    """
    if n == 0 or n == 1:
        return 1
    try:
        return memo[n]
    except KeyError:
        result = fast_fib(n-1, memo) + fast_fib(n-2, memo)
        memo[n] = result
        return result
```

a Python dictionary plays the role of memory.

Dynamic programming

Dynamic programming is an algorithmic paradigm.

- It is a method for efficiently solving problems that exhibit the characteristics of overlapping subproblems and optimal substructure.
  - A problem has overlapping subproblems if it involves solve the same problem multiple times.
  - A problem has optimal substructure if a globally optimal solution can be found by combining optimal solutions to local subproblems.
- It memorizes the results of subproblems to avoid computing the same results again.

Picking the fewest bills

Suppose you want to count out a certain amount of money, say $123, using the fewest bills and coins. (Assume we are in China, we have 7 bills: 100, 50, 20, 10, 5, 2, and 1, and each bill is in sufficient supply.)

Greedy algorithm.

At each step, take the largest possible bill that does not overshoot. So, we will choose:

- a 100 yuan
- a 20 yuan
- a 2 yuan
- a 1 yuan

Dynamic programing: Picking the fewest bills

Suppose, the bills you have is 9, 6, 5, 1, and the value you need is 11. You have infinite supply of each bill. What's the minimum number of bills to get the value?

- If you follow a greedy algorithm, you solution is {9, 1, 1}
- But the optimized solution is {6, 5}

Let's try dynamic programming.

Dynamic programing: Picking the fewest bills

How can we find the bill that minimize the number of bills that count out the current value in each round?

We traverse all bills and compare the corresponding results, as shown in the following pseudo code.

```
bills: a list of all bills available
minBill(value): a function returns the minimum number of bills to
count out the value.

num = inf
idx = 0
for i from 0 to (len(bills)-1):
    if b[i] <= value:
        sub_num = minBill(value-b[i])
        if (sub_num + 1) < num:
           num = sub_num + 1
           idx = i
```

Here, it returns the minimum number of bills of the last round if we pick b[i] in current round.

Recursion: Picking the fewest bills

This is the recursive version based on our previous analysis.

We repeatedly calculate the `(p, sub_num)` for the `(b, value-b)` pair. So, we can keep them in a memo to save time.

```
def minBills(bills, value):
    if value == 0:
        return [], 0
    num = float("inf")
    for b in bills:
        if b <= value:
            p, sub_num = minBills(bills, value-b)
            if sub_num + 1 < num:
                num = sub_num + 1
                pk = p + [b]
    return pk, num


if __name__ == "__main__":

    bills = [9, 6, 5, 1]
    value = 11
    picked, n = minBills(bills, value)
    print(picked, n)
```

Dynamic programing: Picking the fewest bills

The problem can be solved by dynamic programming:

- Substructure: each round, the bill you pick relies on the bills you picked in the past.
- Overlapping: each round, we pick one bill that minimize the number of bills to count out the current value.

DP: Picking the fewest bills

This is the fast_minBills(), very similar to the fast_fib() we written.

```
def fast_minBills(bills, value, memo = {}):

    if value == 0:
        return [], 0

    num = float('inf')
    for b in bills:
        if b <= value:
            try:
                p, sub_num = memo[ value -b]
            except KeyError:
                p, sub_num = fast_minBills(bills, value - b, memo)
                memo[ value-b ] = (p, sub_num)
            if sub_num + 1 < num:
                num = sub_num + 1
                pk = p + [b]
    return pk, num
```

A Python dictionary plays the role of memory.

Steam Sales - Most Valuable First

- What if you just purchase the most valuable games into your inventory?

	Value ($)	Size(GB)
Civilization V	190	10
CS:GO	300	15
The Elder Scrolls V	180	10
Torchlight II	20	2
Age of Empires II	35	5
Banished	10	1

The space limit is 20 GB.

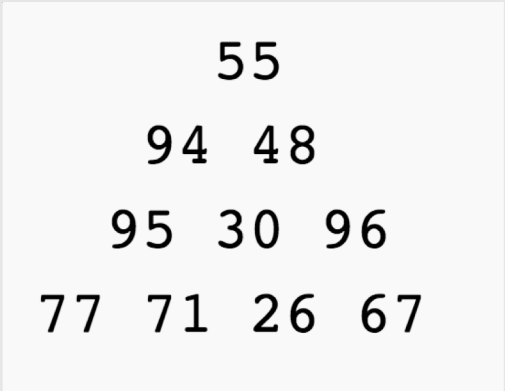
Steam Sales – Most Valuable First

```
def max_val(lst, space):
    if lst == [] or space == 0:
        return [], 0

    # if the first game in lst is too big for the available space
    elif lst[0][1] > space:
        selection, val = max_val(lst[1:], space)
        return selection, val

    # if the first game in lst is eligible for the space
    else:
        """when lst[0] is eligible, there are two senarios:
        1. the best solution contains lst[0]
        2. the best solution does not contain lst[0"""

        # best value with presence of lst[0]


        # best value without presence of lst[1]

        return
```

Algorithm design Workshop

1. Maximum sum descent

- Positive integers in a triangle; a walk from a node can only go to one of the two children nodes
- Goal: find the largest sum of all descents from the root to the base



- One walk is (55 -> 94 -> 30 -> 26) = 205
- Maximum descent is 321 (55 -> 94 -> 95 ->77)

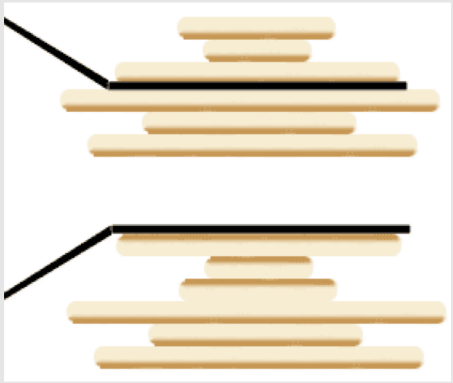
```
In [27]: run maxsum.py
triangle --
[17]
[15, 8]
[5, 10, 8]
[16, 6, 10, 12]
[19, 10, 5, 15, 12]

maximum sum --
[17]
[32, 25]
[37, 42, 33]
[53, 48, 52, 45]
[72, 63, 57, 67, 57]
```

2. Pancake sorting

- n pancakes of different sizes, randomly stacked
- Allowed action: slip a spatula under one pancake, and flip
- Goal: sort the pancakes (smallest at the top)

The following code is for generation of “pancake”.

```
import random
random.seed(0)


def get_pancake(a, b, n):
    return [random.randint(a, b) for i in range(n)]
```



Let a = [1, 4, 2, 6, 3], in pancake sorting, you are only allowed to change the position of an element by slip-flip. For example, if we want to move 6, we can only move it to the beginning of a, and, after the moving, a = [6, 2, 4, 1, 3]. We can move 6 to the end of a by another slip-flip, then, a = [3, 1, 4, 2, 6].

3. Number placement

- n numbers; n-1 preset inequality sign
- Goal: insert the numbers so that the inequality holds

Example:

Numbers: [2, 3, 0, 1, 5]; Signs: [‘<’, ‘>’, ‘<’, ‘<’]

Solution: 0 < 5 > 1 < 2 <3.

```
In [35]: run sign_ins.py
[1, '<', 20, '>', 9, '<', 19, '>', 16, '>', 10, '<', 13, '>', 12]
```

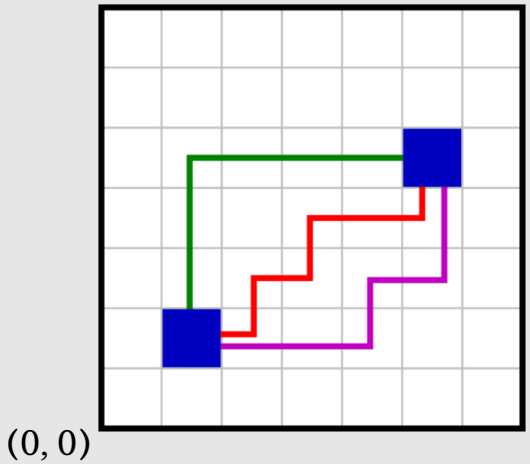
4. Site selection

Goal: set up an ice cream stand; minimize the travel distances for your friends

- n coordinate pairs (x_1, y_1) ... (x_n, y_n) on an integer grid
- Pick (x, y) s.t. sum($|x - x_1| + |y - y_1| + |x - x_2| + |y - y_2| + \dots$) is minimum.

You can randomly generate a set of coordinates by the following.

```
import random
random.seed(0) # optional

def get_coordinates(n):
    x = [i for i in range(n)]
    y = [i for i in range(n)]
    random.shuffle(x)
    random.shuffle(y)
    return [c for c in zip(x, y)]
```

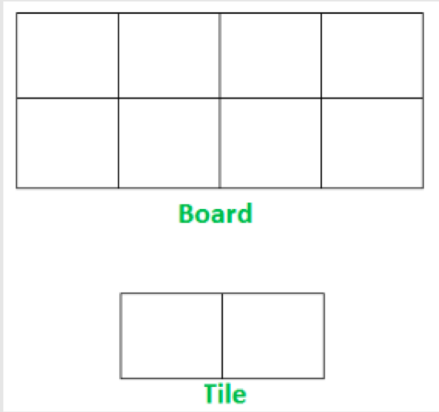


Manhattan distance:
For two points (x_1, y_1), (x_2, y_2), the Manhattan distance is $|x_1-x_2|+|y_1+ y_2|$.

The lower corner is (0, 0); the two blue points is (1, 1) and (5, 4).

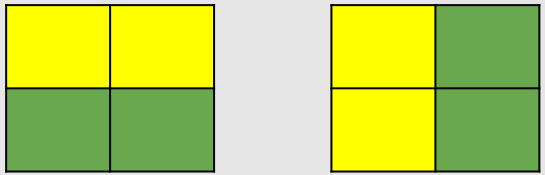
5. Tiling problem

- Given a “2 × n” board and tiles size “2 × 1”, count the number of ways to tile the given board using the 2 × 1 tiles.
- A tile can either be placed horizontally, i.e., as a 1 × 2 tile or vertically, i.e., as 2 × 1 tile.



When n=2, we have two ways to cover the board:

- Two tiles in horizontal
- Two tiles in vertical



6. Anagram detection

Anagram: words with same letters (e.g., “eat”, “ate”, “tea”)

Finding all anagram words.

The starting code and a word list is given in the NYU Class Resources folder
Hint: you may need a Python package named itertools.

```
from itertools import permutations
```

```
def get_anagrams(word,vocabulary):
    # write your code here
    pass

if __name__ == "__main__":
    # Read english.txt, create list of words
    all_words = open("english.txt","r").read().split("\n")
    # Read input.txt, get list of test cases
    input_list = open("input.txt","r").read().split("\n")

    for word in input_list:
        result = get_anagrams(word, all_words)
        print(word,":",result)


"""
TEST set answers:

cater : ['carte', 'caret', 'crate', 'trace', 'recta', 'react']
race : ['acre', 'care']
pants : []
tea : ['eta', 'eat', 'ate']
sound : ['nodus']
ring : ['grin']
road : []
"""
```