Lecture 6.1

Algorithm Design Strategies I

Exam Date and Instructions

- Exam date: 19/04/2023 – 5:50 PM to 8:15 PM (Melb time)

- Exam instructions: please check LMS announcements and the attached file.

Topics 6.1 and 6.2 Intended Learning Outcomes

- By the end of the week you should be able to:
  - Compare the time efficiencies of different algorithms,
  - Understand the difference between exact and approximate solutions,
  - Apply generic sorting and searching algorithms to solve certain problems, and
  - Use recursion to solve problems in a divide-and-conquer fashion.

Lecture Overview

1. Approaching Problems
2. Algorithm Complexity
3. Approximate Solutions

Approaching Problems

Scary New Problems 👻

- Approaching a new programming problem can be daunting.
  - The algorithm for the solution may not be immediately **obvious**.

- Often a lot of the "**scariness**" comes from trying to consider **everything** all at once and becoming overwhelmed.

Key Strategies for New Problems

- Fortunately there are strategies which can help you get a foothold and start designing an algorithm:

1. Break up the problem into smaller, more manageable sub-problems.
2. Transform the (sub-)problem into another problem that you already know how to solve.
3. Create traces by solving the (sub-)problem for different data manually.

Breaking Up The Problem

- Real-world problems can often be broken up into several **smaller**, **easier** problems.

- Try to look for parts of a task description which can be solved **independently** of each other.

- Once you've **solved** these **parts**, consider the problem as a **whole** and use those solutions as **building** blocks to solve the overall problem.

Breaking Up The Problem

- For example, consider the problem we solved in an earlier lecture: **finding the most frequent word**.

- We solved this by breaking up the problem into two sub-problems:
  - **Count all of the words**, and
  - **Find the highest word count from all word counts**.

Breaking Up The Problem

- Sometimes breaking up a problem can involve finding a **self-similar** problem which is in some way **easier** to solve than the **original** problem.
- This leads to a technique known as recursion.
- We will discuss recursion in the next lecture.

Transforming Problems

- A large part of algorithm design is recognising **problem patterns**.
- This allows us to **transform specific** problems into **generic** problems that we **know** the solution to.
- Being able to **spot** patterns is a real programming **skill** that **improves** with **experience**.

Example: Transforming Problems

- Task: "Find the age of the oldest user".
- This problem fits a pattern we've encountered: minimum/maximum aggregation.
  - Given a list of user **ages**, find the maximum value.

- What if the users are stored with their date of **birth** instead of numerical **ages**?
  - No problem, just break up the problem: first **calculate** user ages, then **find** the maximum age.

Transforming Problems

- It won't always be possible to transform a specific problem into a generic **equivalent precisely**.
- However, you can often find a **generic** problem which is *close*, which can give clues about how to solve your problem.
- Sometimes you have to get a bit **creative** with your problem solving!

Transforming Problems

- **Common** generic problems which appear repeatedly during program are:
  - **Sorting**, and
  - **Searching**.
- We will discuss sorting and searching in detail next lecture.
- **Aggregation**, **mapping**, and **filtering** are also extremely common components of solutions.

Creating Traces

- If you are struggling to **break up** the problem or recognise a **generic pattern**, you can try working through the solution steps manually.
- Select some data values for inputs and try to "think like a computer" as you calculate the outputs by hand.
- Explicitly write down each step you take, and don't skip over things that you consider "obvious".
  - Pretend that you are explaining the process to a **baby robot**.

Creating Traces

- Once you've created a few traces, try identifying important elements like decisions and process steps.
- Assemble the elements into a flowchart.
- If you're lucky, the act of creating a trace might reveal a problem pattern you missed previously.

Creating Traces

- Creating traces is also helpful for creating test cases.
- Use the input/output pairs you worked through as test examples.
- If your program gives different results to what you expect, you can dive into debugging.

Algorithm Complexity

Time Complexity

- Often many **different** possible algorithms solve the **same** problem correctly.
- However, some solutions may be **faster** than others.
- To compare how **efficient** algorithms are, we compare their time complexity.
- Time complexity gives a **rough** indication of **how** many computer **instructions** must be executed in order to arrive at a result.

Time Complexity

- In general, lower time complexity is better.
  - Quicker results and lower hardware requirements.

- Some algorithms are *reaaaaaalllllly* slow, to the point of being impractical.
- There's not much use for a program which literally **takes** 10,000 years to produce an output!
- For this reason a **basic** understanding of complexity analysis is useful for identifying when otherwise correct algorithms are not good solutions.

Complexity Analysis

- Complexity analysis is an in-depth area of study within computer science.
- It involves describing the **time** (and **space**) complexity of an algorithm.
  - **Time complexity**: How much time a program takes to run.
  - **Space complexity**: How much space (memory) a program requires.
- We will focus on **time** complexity analysis.

Complexity Analysis

- The complexity of an algorithm is usually expressed in terms of **its input**.

- Therefore the complexity **describes** how the required time/space **grows** as a **function** of the input.

- Generally **bigger input** means **slower execution**.
  - Complexity analysis tells us how much **slower** things get as the input gets **bigger** (i.e. what is the trend?).

Worst-Case Complexity

- **Different** inputs of the same **size** may take different amounts of time to process.

- We are usually concerned with worst-case time complexity.
  - i.e. how long it will take to run in the "**unluckiest**" scenario.

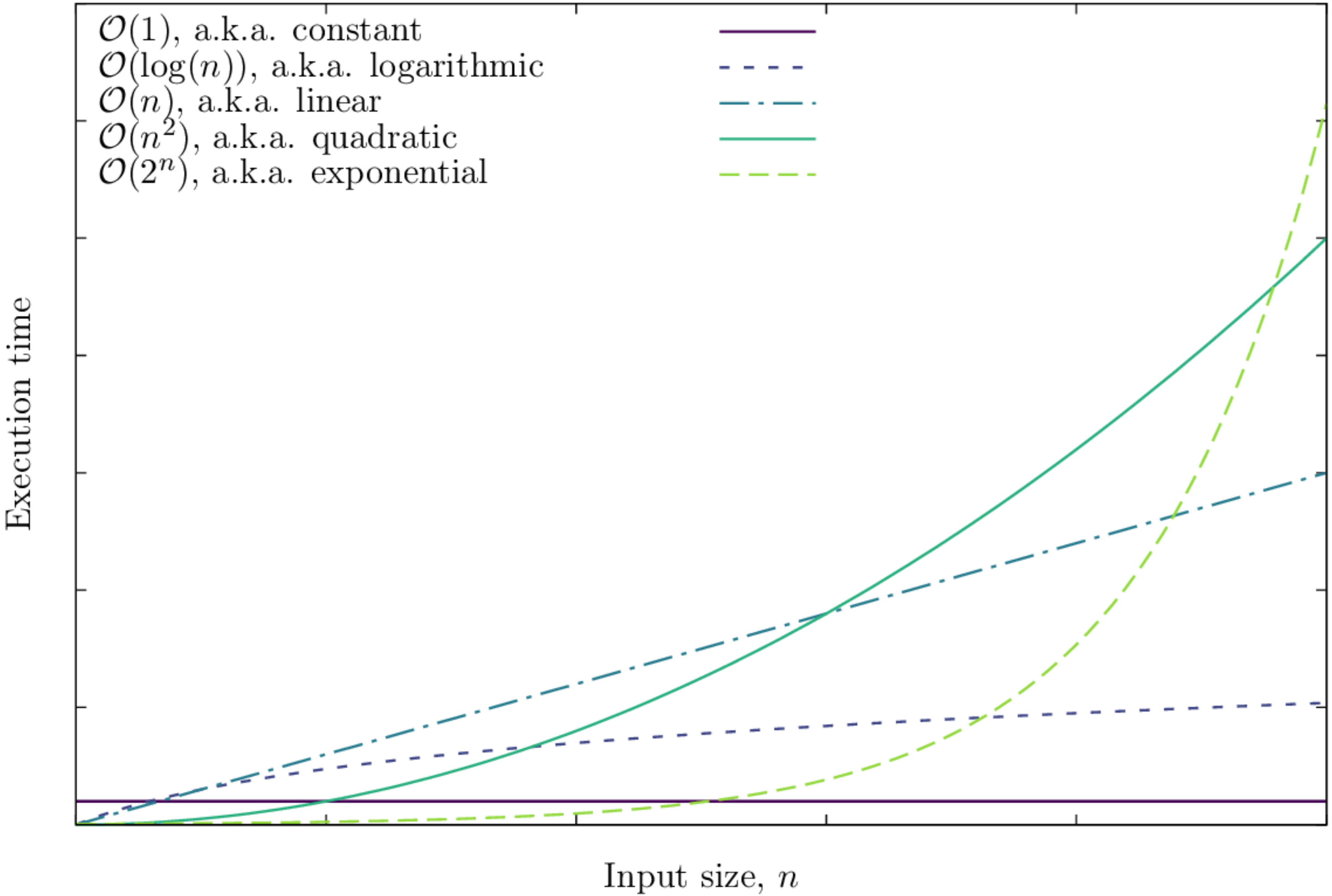
- For example, it might take longer to **sort** a **scrambled** list than a **list** that is **already** sorted.

Worst-Case Complexity

- Worst-case complexity is expressed using big O notation.

- **Big O** notation is not an exact formula for calculating the execution time, but expresses the general relationship between **input size** and **time**.

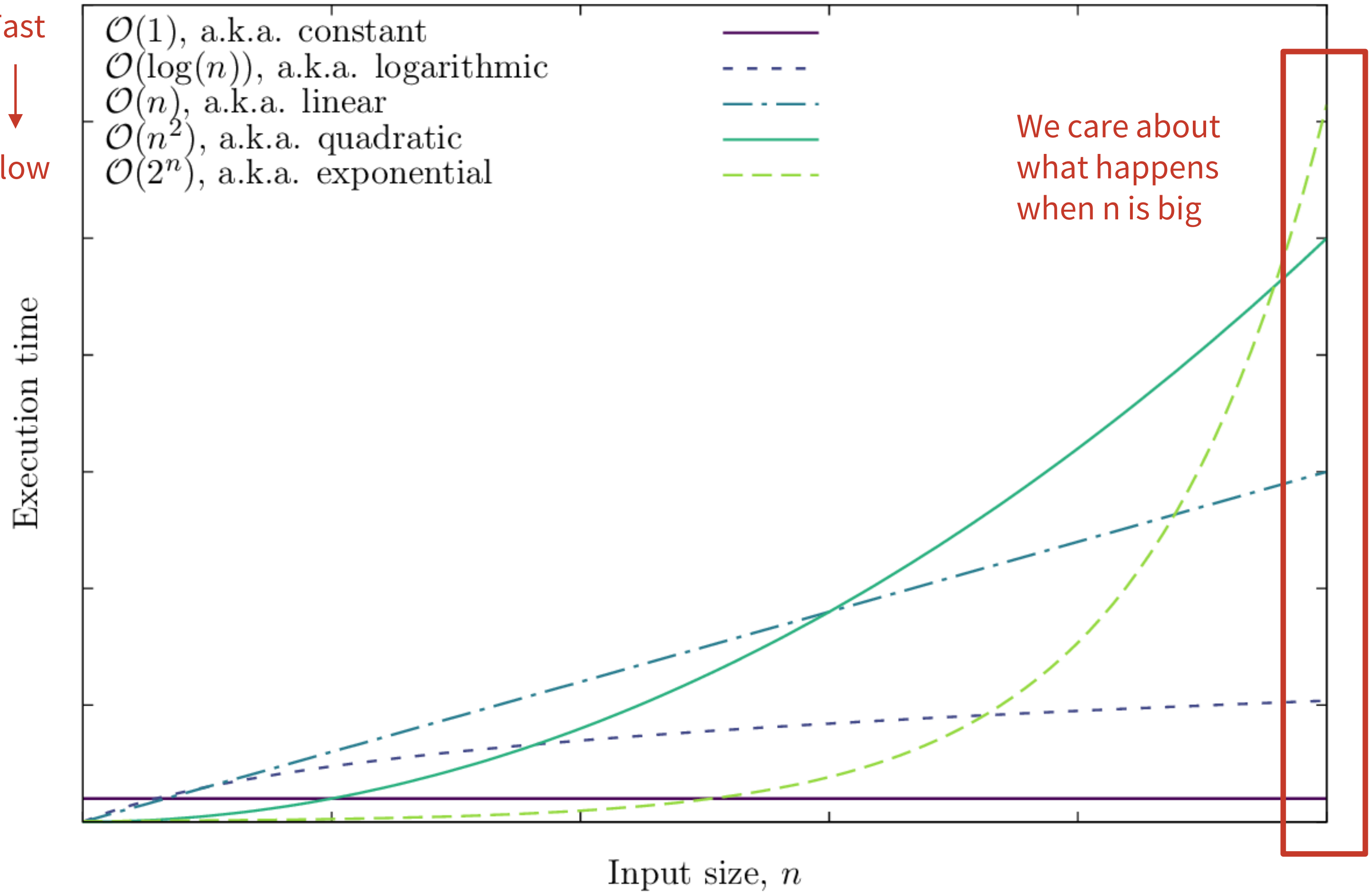
- What we care about is **how slow the program** becomes as the **input size**, **n**, becomes **large**.


$\mathcal{O}(1)$, a.k.a. constant
$\mathcal{O}(\log(n))$, a.k.a. logarithmic
$\mathcal{O}(n)$, a.k.a. linear
$\mathcal{O}(n^2)$, a.k.a. quadratic
$\mathcal{O}(2^n)$, a.k.a. exponential
Execution time
Input size, n

Fast

↓

Slow



We care about what happens when n is big

Comparing Complexity

- We care about **differences** as the input gets **large**.
- An $O(n)$ and an $O(n^2)$ algorithm will be roughly equivalent in speed when $n = 1$.
- However, the $O(n)$ algorithm will be much faster when ***n*** gets large.
  - This is what we care about in complexity analysis.
- So we consider **$O(n)$** algorithms to be faster than **$O(n^2)$** algorithms.

Example: Linear Time

- An O(n), or "linear-time" program performs a number of steps proportional to the input size.
    - e.g. doubling each number in a list.
- Linear-time programs usually contain a loop (but **no nested loops**).

```python
n = int(input("Enter a number: "))
total = 0
for x in range(1, n + 1):
    total = total + x

print(total)
```

- The above program is linear-time.
- The number of statements executed **grows proportionally** to the value of ***n***.

Example: Constant Time

- An O(1), or "constant-time" program performs a **fixed number of steps** regardless of the **input size**.
  - e.g. calculating BMI from height and weight values.
- **Constant-time** programs usually contain **no loops**.
- Constant-time programs are the fastest.

```
n = int(input("Enter a number: "))
total = (n * (n + 1)) // 2

print(total)
```

- The above program is constant-time.
- It is the functionally the same as the previous program, but faster.
- There are multiple ways to skin a cat, but some ways are more efficient 🙀

Brute Force Solutions

- Often the most obvious algorithm for solving a problem is the brute force solution.
    - No, *this does not involve forcefully embedding your keyboard in the LCD panel.*

- A brute force solution is an **exhaustive** solution which, in some sense, **searches** through **every possibility** to find a result.

Example: Combination Locks

- Consider a **4**-digit combination lock.
- The brute force solution to opening the lock is **trying every combination**.
- In the worst case, we get it on the last try which would take **10 x 10 x 10 x 10** = 10^4 attempts.

- Complexity for a lock with n digits: $O(10^n)$
  - This is exponential time.
  - Equivalent complexity to $\mathbf{O(2^n)}$.

Example: Combination Locks

- As an aside, this exponential time complexity is why long passwords are important.
  - Each **additional character** ***multiplies*** the **time** taken to guess the password.

- If you **know** the **combination** for a lock, then the **complexity** for opening it **linear-time**, O(n).
  - You still need to enter each of the ***n*** digits.

- As you can see, the difference between **O(n)** and **O(2^n)** is very **significant**.

Brute Force Solutions

- For some applications the brute force solution actually can be reasonable (typically when **inputs are small**).

- For many others, the brute force solution is **impractical**, especially when the **solution** is **exponential-time**.
    - In such cases it is necessary to **think** about the problem more and develop a more **efficient** algorithm.

Example: Cheapest Pair of Items

Task description

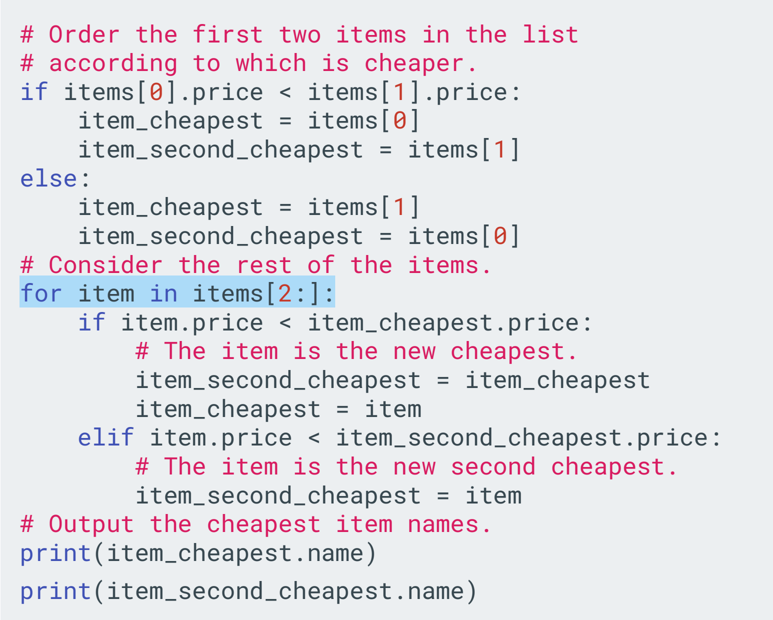
Given a list of items in a store, find the pair of unique items with the lowest total purchase price.

Example: Brute Force Solution

```
best_pair = []
best_price = 999999
# Nested loops consider every pair of items.
for i in range(len(items)):
    for j in range(len(items)):
        # If both items are the same, skip.
        if i == j:
            continue
        # Compare pair price to the current best.
        price = items[i].price + items[j].price
        if price < best_price:
            best_price = price
            best_pair = [items[i], items[j]]
# Output the item names.
print(best_pair[0].name)
print(best_pair[1].name)
```

- Let's call the length of items ***n***.
- This solution considers all combinations of items.
    - **n x n = n^2**
- Hence the time complexity for this solution is **O(n^2)**, or "**quadratic**".

Example: Faster Solution



```
# Order the first two items in the list
# according to which is cheaper.
if items[0].price < items[1].price:
    item_cheapest = items[0]
    item_second_cheapest = items[1]
else:
    item_cheapest = items[1]
    item_second_cheapest = items[0]
# Consider the rest of the items.
for item in items[2:]:
    if item.price < item_cheapest.price:
        # The item is the new cheapest.
        item_second_cheapest = item_cheapest
        item_cheapest = item
    elif item.price < item_second_cheapest.price:
        # The item is the new second cheapest.
        item_second_cheapest = item
# Output the cheapest item names.
print(item_cheapest.name)
print(item_second_cheapest.name)
```

- Finding the cheapest pair of items is equivalent to finding the cheapest two items.
- This solution only needs to consider each item in items once (no nested loop).
- This solution is **O(n)**, or "linear".
- Since **n** < **n^2** for large **n**, this solution is considered faster.

Approximate Solutions

Optimal and Approximate Solutions

- A solution is optimal if it gives the best possible answer for a problem.
- That is, it always solves the problem perfectly.

- A solution is approximate if, in some scenarios, it produces results which are in some way imperfect or sub-optimal.
- The results might still be close to optimal.

Optimality and Speed

- It is an **uncomfortable** truth that you will **not** always be able to solve a problem both **optimally** and **practically**.
  - In such cases, you may have no choice but to use an **approximate** solution.

- A famous example is the *travelling salesperson problem*.

Travelling Salesperson Problem

Task description

Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

—Wikipedia

- This problem is actually really hard!
- No known way to return an optimal answer in reasonable time for a large number of cities.
- Can take a "near enough is good enough approach".
  - i.e. find a good route quickly that is not necessarily the best.

Heuristics

- Techniques employed in algorithms which are **not** guaranteed to be **optimal** are called heuristics.

- Heuristics help us arrive at solutions that aren't optimal, but are **good enough**.
  - Sacrifice **optimality** for a **reasonable** running **time**.

- Since sometimes heuristics are **required for a practical** solution, it is important to be **aware** of them.

Greedy Algorithms

- A greedy algorithm is a heuristic approach to solving a problem.

- It involves **repeatedly** making decisions which are locally **optimal**, but are not **guaranteed** to lead to a **globally** optimal solution.
    - i.e. greedy algorithms make "short-sighted" decisions, selecting what is **best** at a **given step** without considering **future implications**.

Example: Dividing Loot

Task description

Alice and Bob are famous explorers who frequently discover bags full of precious gemstones. Write a program which divides a bag of gemstones as evenly as possible between the two explorers, based on their values.

Example: Dividing Loot

- To simplify the problem, we will assume that each gemstone is represented by a single number (its value).
  - Our program input is a list of numbers.

Dividing Loot with a Greedy Solution

```
def divide_gems_greedy(gems):
    # These two lists represent the gems assigned to
    # Alice and Bob.
    gems_a = []
    gems_b = []
    # Iterate over the gems.
    for gem in gems:
        # If Alice holds lower total value, give the gem
        # to her. Otherwise give it to Bob.
        if sum(gems_a) <= sum(gems_b):
            gems_a.append(gem)
        else:
            gems_b.append(gem)
    # Return the results.
    return gems_a, gems_b
```

Dividing Loot with a Greedy Solution

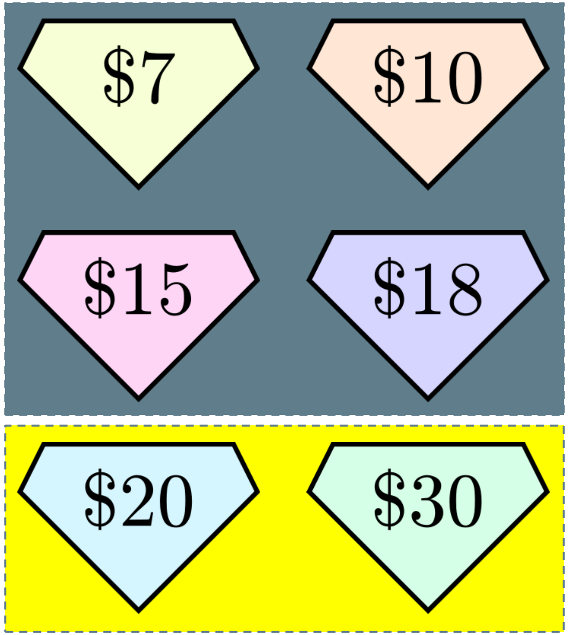
- This algorithm is greedy because it considers one gem at a time without considering implications for the remaining gems.
- The greedy algorithm is relatively fast (linear-time).
- However, it is not optimal.
    - i.e. there may be a better way to balance the gemstones than the greedy algorithm suggests.

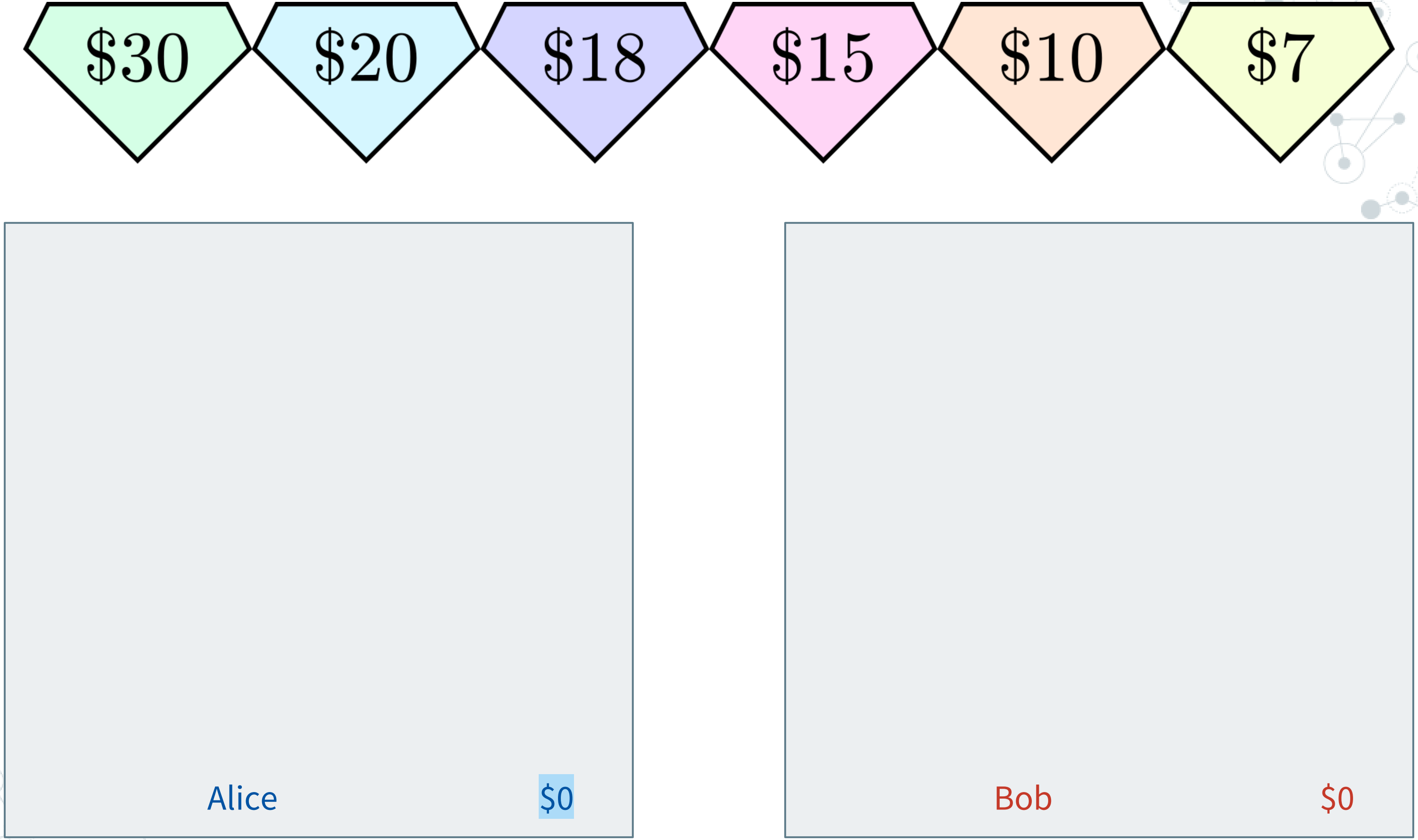
Dividing Loot with a Greedy Solution

- Let's say that we have the gems shown here.

- The optimal way of dividing them is as follows:
  - $20 + $30 = $**50**
  - $18 + $15 + $10 + $7 = $**50**

- However, the greedy solution does not arrive at this answer.

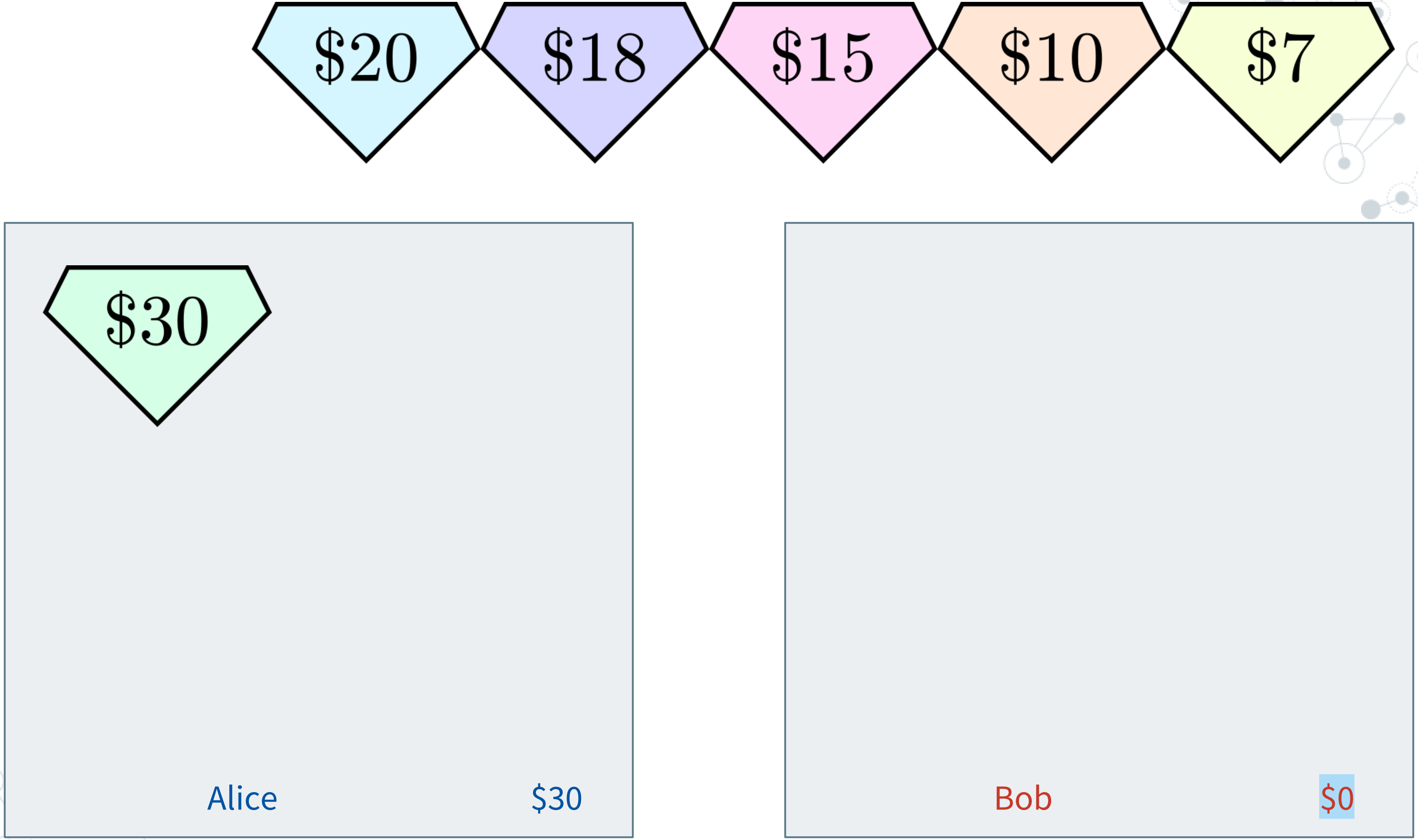




$30 $20 $18 $15 $10 $7

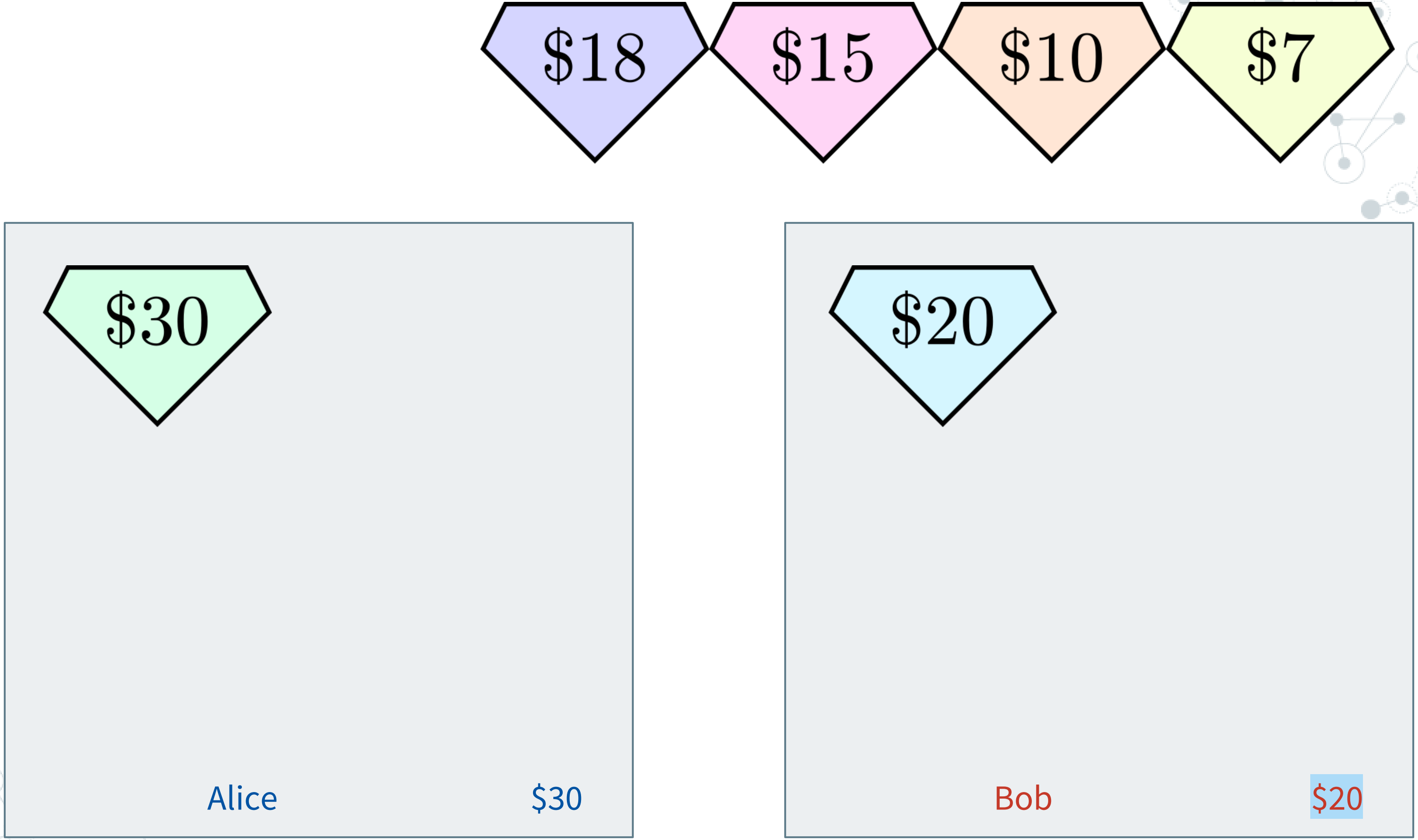
Alice $0

Bob $0



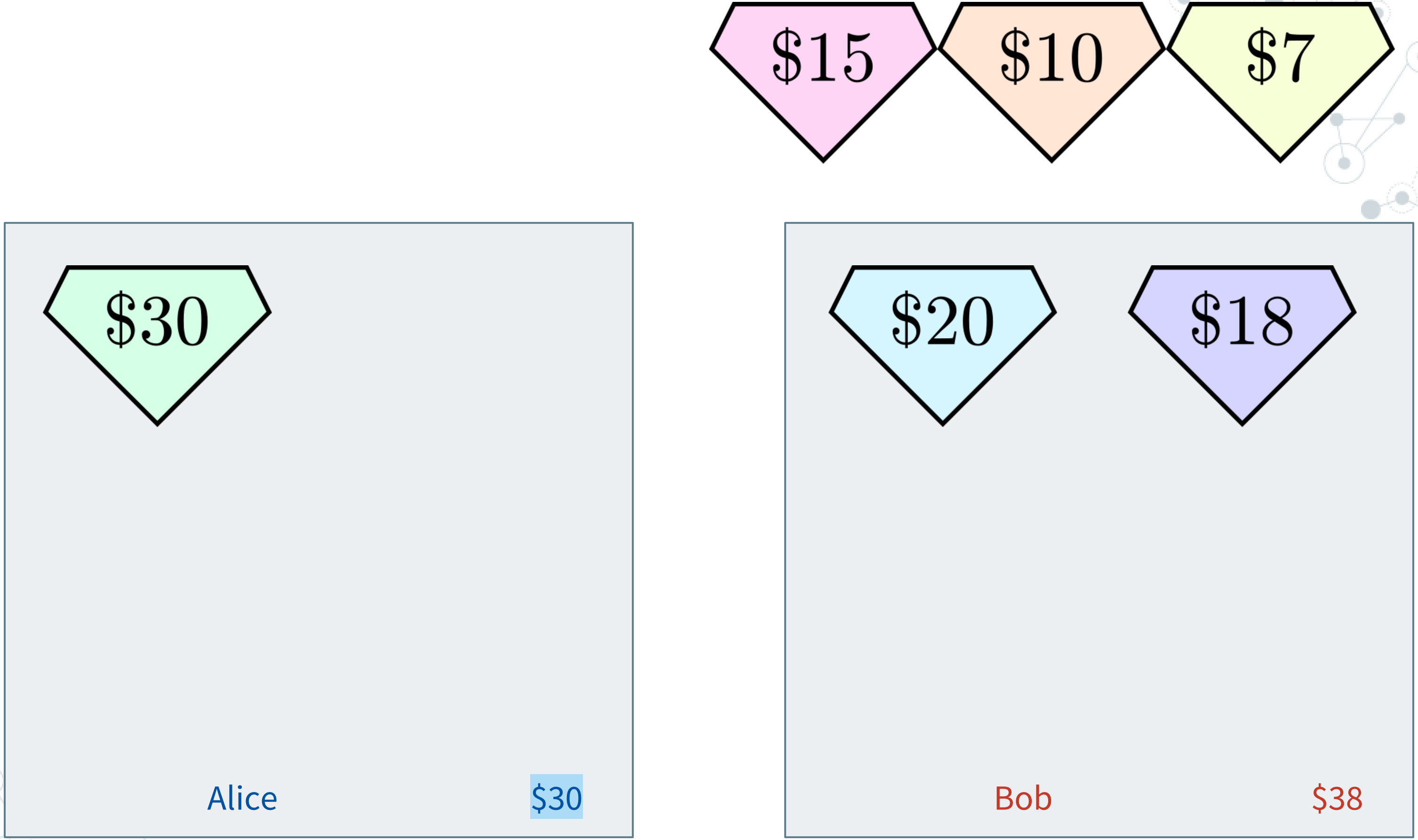
Alice $30

Bob $0



Alice $30

Bob $20



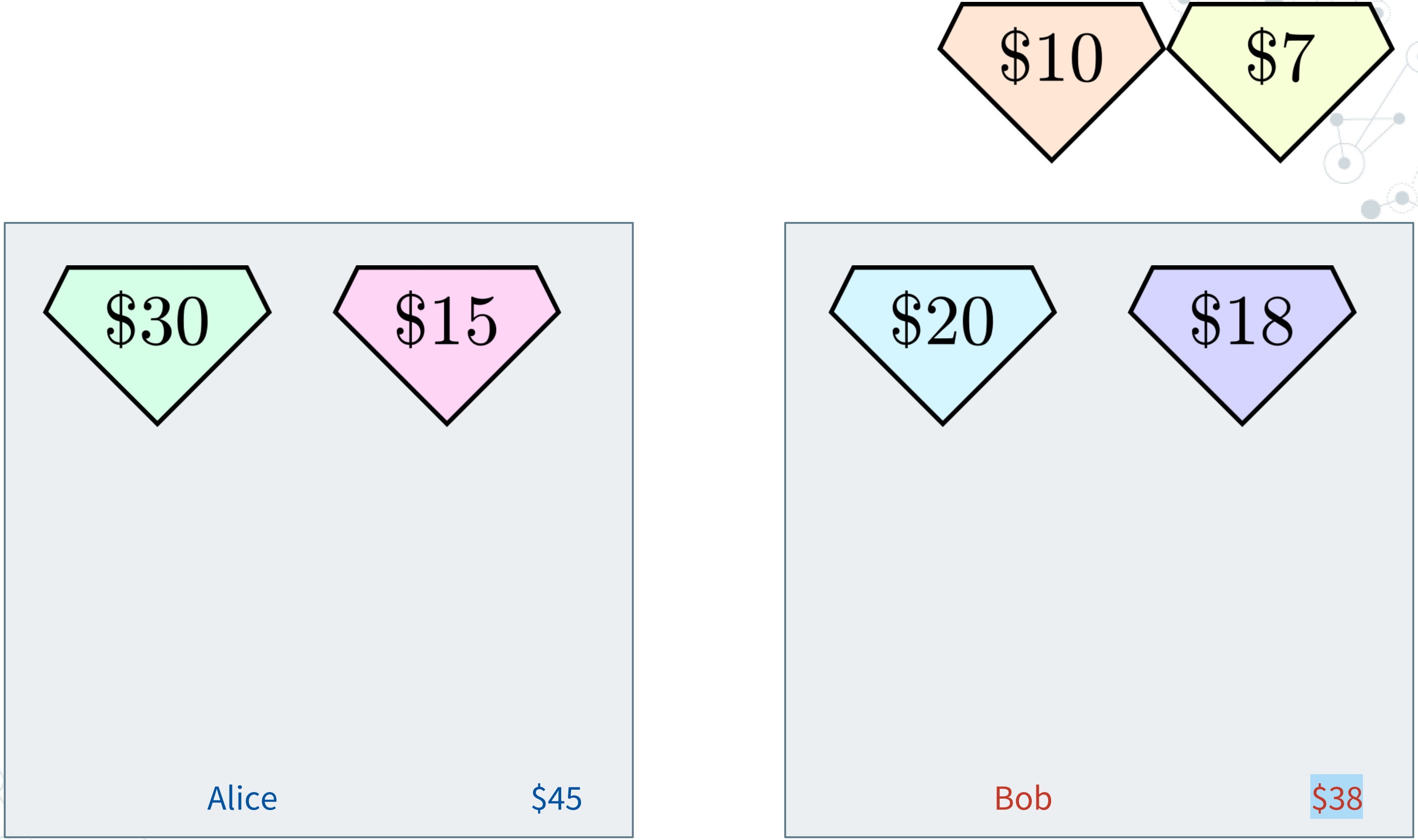
$15 $10 $7

$30

$20 $18

Alice $30

Bob $38



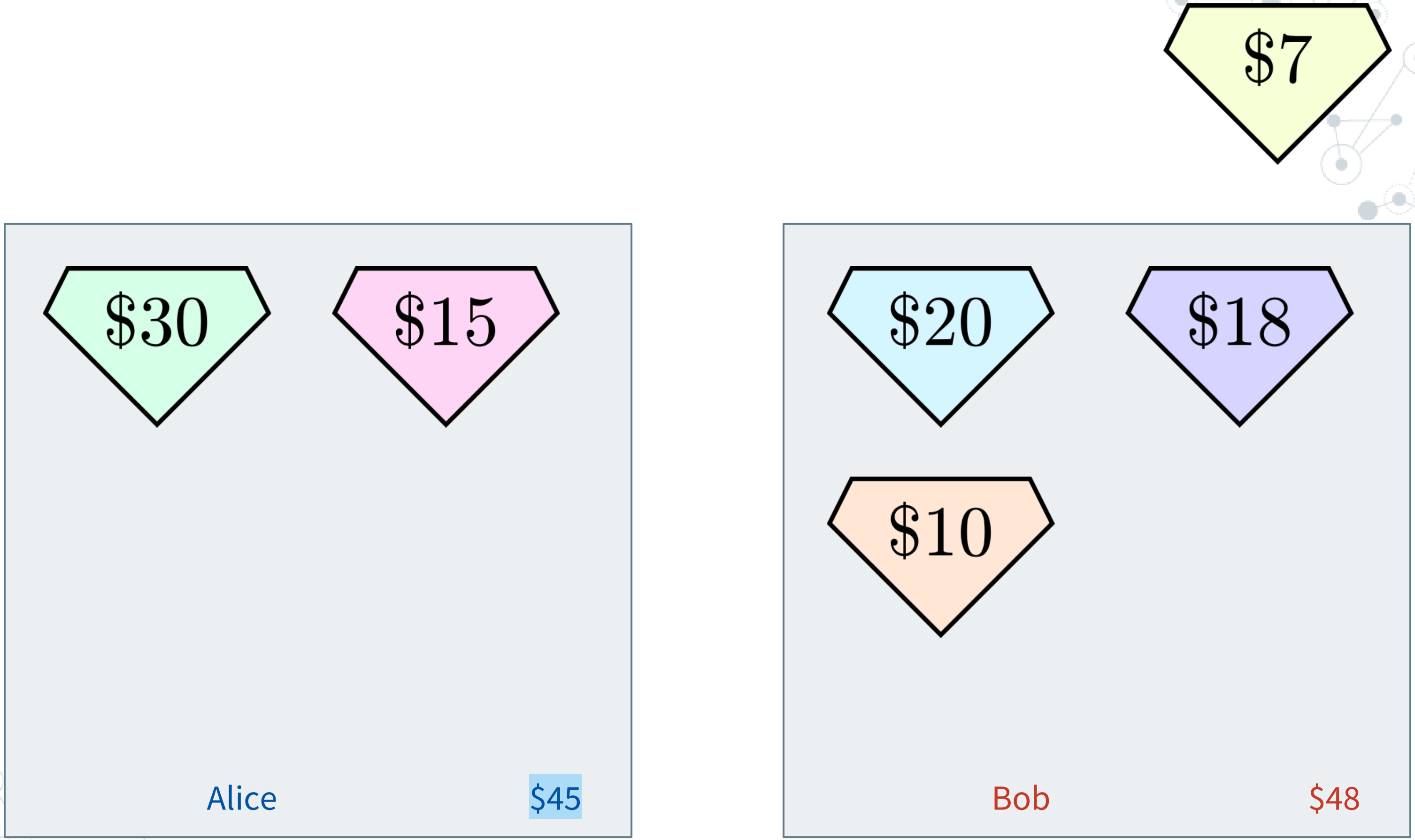
$10 $7

$30 $15

Alice $45

$20 $18

Bob $38



$7

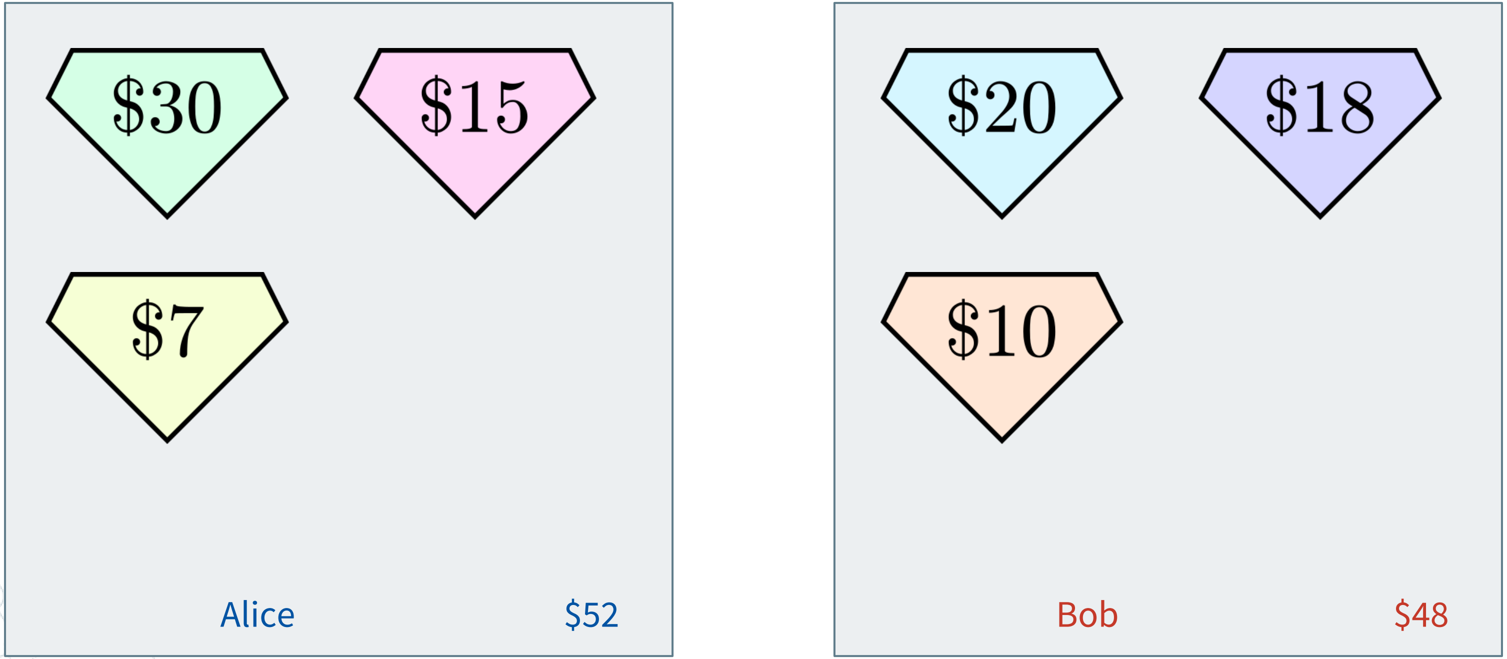
$30 $15

Alice $45

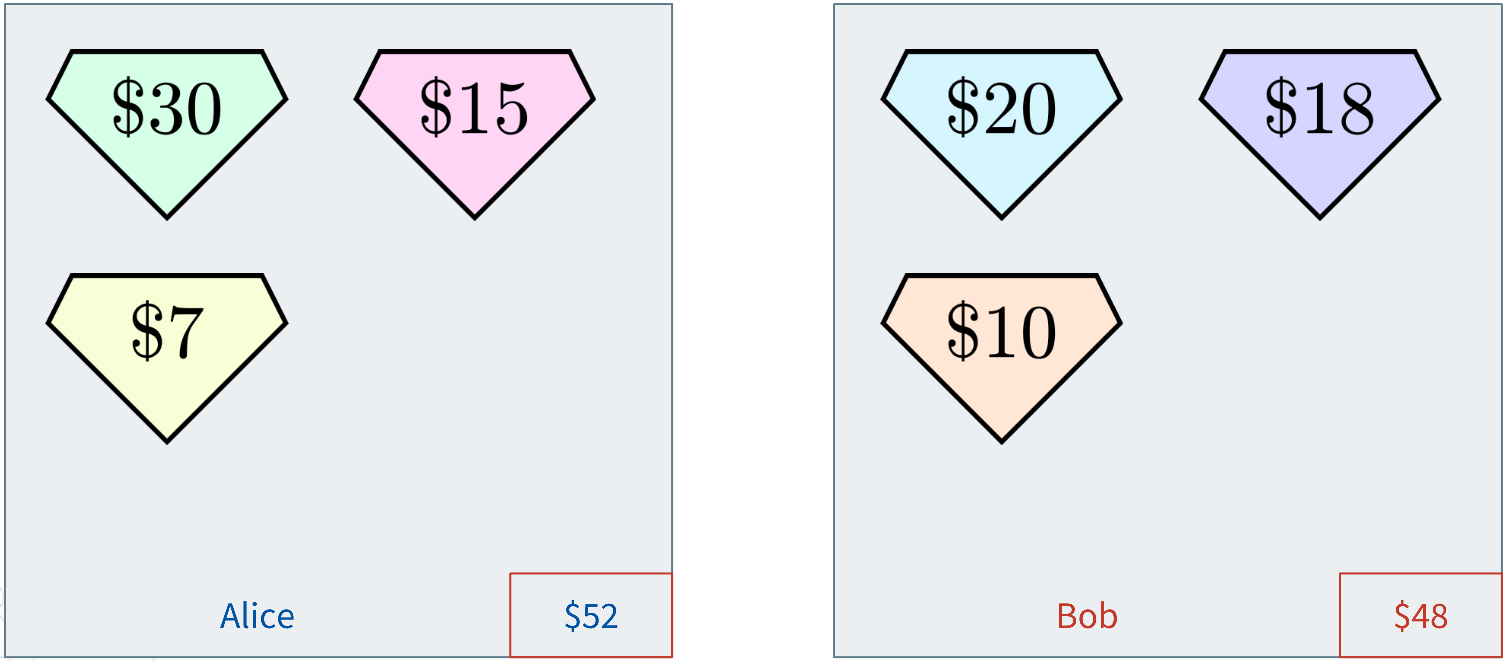
$20 $18

$10

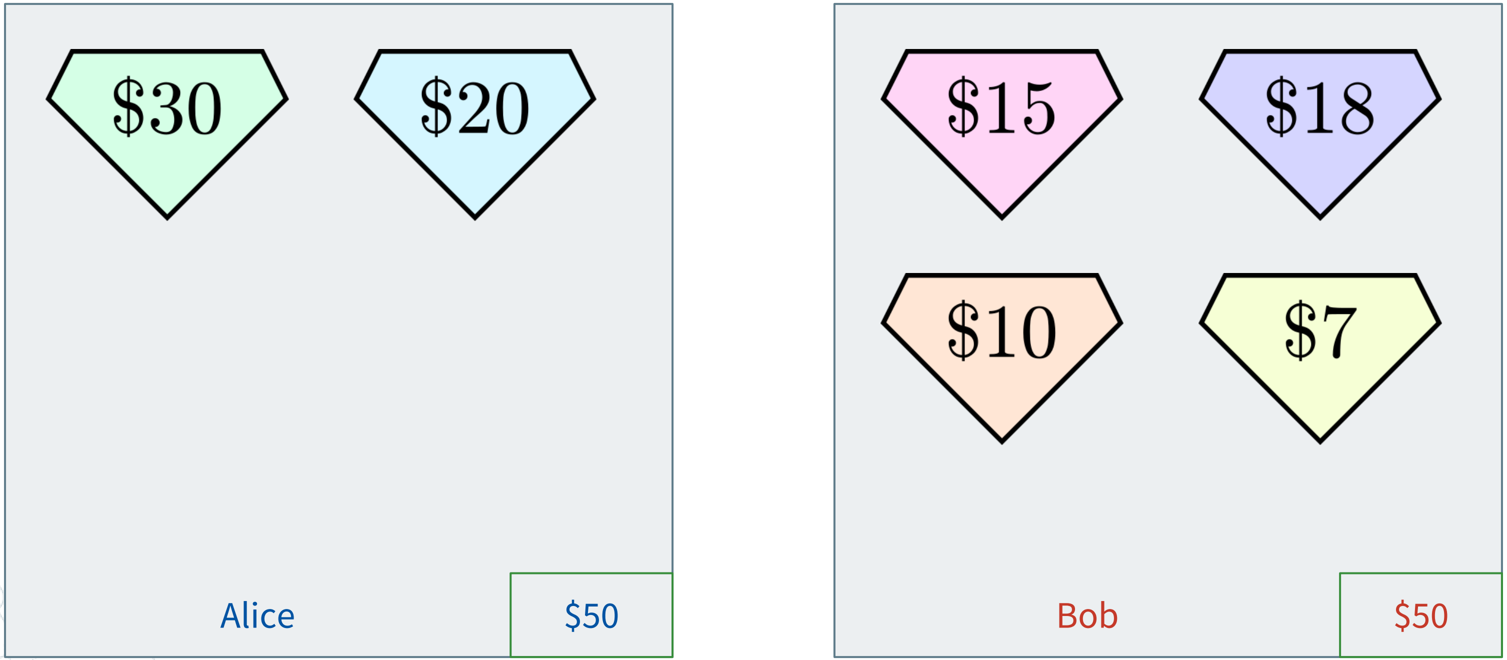
Bob $48


$30
$15
$7
Alice
$52
$20
$18
$10
Bob
$48

Not optimal! Recall that we can achieve a split of $50 each.



We should have reached the solution below.



$30

$20

Alice

$50

$15

$18

$10

$7

Bob

$50

Dividing Loot with a Greedy Solution

- Even though we are making locally optimal decisions, we missed out on the **best** overall solution.

- It's important to consider whether the loss of **optimality** from taking a **heuristic** approach is acceptable or not when developing an algorithm.

- We will return to this problem **next** lecture.
  - We will find an optimal (but slow) algorithm.

Check Your Understanding

Q. The greedy solution of the gem-dividing problem presented here calculates the total sum of values at each step. How could you improve the efficiency of the algorithm by avoiding this?

```
def divide_gems_greedy(gems):
    gems_a = []
    gems_b = []
    for gem in gems:
        if sum(gems_a) <= sum(gems_b):
            gems_a.append(gem)
        else:
            gems_b.append(gem)
    return gems_a, gems_b
```

Check Your Understanding

Q. The greedy solution of the gem-dividing problem presented here calculates the total sum of values at each step. How could you improve the efficiency of the algorithm by avoiding this?

A. By keeping track of the sums as we go using two extra variables, as shown.

```python
def divide_gems_greedy(gems):
    gems_a = []
    gems_b = []
    # Initial sums are zero.
    sum_a = 0
    sum_b = 0
    for gem in gems:
        if sum_a <= sum_b:
            gems_a.append(gem)
            # Add to Alice's total.
            sum_a = sum_a + gem
        else:
            gems_b.append(gem)
            # Add to Bob's total.
            sum_b = sum_b + gem

    return gems_a, gems_b
```

Lecture 6.2

Algorithm Design Strategies II

Lecture Overview

1. Sorting
2. Searching
3. Recursion

Sorting

Sorting as a Solution

- There are certain algorithms which often form a key role in programming problem solutions.
- One example is sorting, which we will discuss now.
  - There are many cases when simply sorting data makes the solution to a problem evident.
- Sometimes it is not immediately obvious that sorting can help.
  - That's why it is useful to always have sorting in your "bag of tricks" for you to consider.

Example: Anagrams

Task description

An anagram of a word is another word with the letters rearranged. For example, “listen” is an anagram of “silent”. Write a program that, when given two words, determines whether they are anagrams.

Example: Anagrams

- Believe it or not, this problem can be solved very simply by sorting!

- Algorithm:
  1. Sort the letters within each word.
  2. If the sorted words are the same, then they are anagrams. Otherwise, they are not.

Example: Anagrams

Input: "silent", "listen"

Sorted: "eilnst", "eilnst"

The sorted versions of the words are equal.

Therefore the two input words are anagrams.

Input: "silent", "listens"

Sorted: "eilnst", "eilnsst"

The sorted versions of the words are not equal.

Therefore the two input words are not anagrams.

Example: Anagrams

```python
def is_anagram(word1, word2):
    # Convert the strings into
    # lists of characters.
    # This is necessary to use
    # the `sort` list method.
    chars1 = list(word1)
    chars2 = list(word2)
    # Sort the characters.
    chars1.sort()
    chars2.sort()
    # Check whether the sorted
    # characters are equal.
    return chars1 == chars2
```

- Here is a function which implements our algorithm.
- Note how simple the code is since we can use the sort list method to do most of the work.
- If you're still not convinced that the algorithm works, try a few examples yourself.

Sorting by Key

- We know that a list can be sorted using the `sort` **list** method.

- This works well when **items** in the **list** are of a similar **type** that can only really be compared in one way.
    - e.g. numbers or strings.

Sorting by Key

- The sort method can also be used on more complex lists, like lists of dictionaries or custom objects.

- To do this, you can specify your own function which tells Python what to base the ordering on.

- This is very useful when devising algorithms based on sorting.

Example: Sorting People

- Let's say that we have the following Person class:

```
class Person:
    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age

    def __str__(self):
        return f'{self.first_name} {self.last_name}, {self.age}'
```

Example: Sorting People

- As part of our Person class we defined a __str__ method.
- This method tells Python how to convert an instance of that class into a string (via the `str` function).
  - Useful for debugging purposes.
- In this case, we've specified that a Person object should be represented as a string in the format 'First Last, Age'
  - e.g. John Doe, 42

Example: Sorting People

- We can create a list of Person object instances.

```
people = [
    Person('Trent', 'Reznor', 55),
    Person('Atticus', 'Ross', 52),
    Person('Alessandro', 'Cortini', 44),
    Person('Chris', 'Vrenna', 53),
    Person('Charlie', 'Clouser', 57),
]
```

Example: Sorting People

- Now suppose that we want to sort these people **by age**.
- We cannot simply call the sort list method.

```
>>> people.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'Person' and 'Person'
```

Example: Sorting People

```
>>> people.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'Person' and 'Person'
```

- The above code does not work because we have not told Python how to compare two Person objects.

- Should they be sorted by first name? Last name? Age? We haven't specified.

Example: Sorting People

```
>>> people.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'Person' and 'Person'
```

- We can clarify our intent in Python by passing an additional argument to the sort list method.
  - This argument is a function which maps an item in the list to a "**key**" used for sorting.

Example: Sorting People

- We'll start by defining a function which maps a person to their age.

```
def get_age(person):
    return person.age
```

- Now we can use the key named argument to tell Python that we want to sort by age.

```
>>> people.sort(key=get_age)
```

Example: Sorting People

- Now the sort method will map people to their ages when deciding which person should appear first in the list.
- This mapping is for sorting purposes only---the actual list items won't be changed.
- We can verify this by printing out the people after sorting.

```
>>> for person in people:
...     print(str(person))
...
Alessandro Cortini, 44
Atticus Ross, 52
Chris Vrenna, 53
Trent Reznor, 55

Charlie Clouser, 57
```

- Each person is printed out nicely thanks to our __str__ method.

Check Your Understanding

Q. How would we modify the code to sort people by last name instead of age?

Check Your Understanding

Q. How would we modify the code to sort people by last name instead of age?

A. We define a new function which maps a person to their last name, and then use this function as the `key` named argument in `sort`.

```
def get_last_name(person):
    return person.last_name


people.sort(key=get_last_name)
```

Searching

Searching as a Solution

- Searching for an item in a list is another basic algorithm that forms an important part of many solutions.
- Sometimes this is obvious.
  - e.g. Searching for a **word** in a text **document**.
- In other cases it will be less **obvious**.
- Like **sorting**, **searching** is an important algorithmic design tool in a programmer's arsenal.

Linear Search

- One way of searching for an item is to consider each item one at a time.
- For each item, decide whether it is the one that we're looking for.
- This is called linear search.
- As the name implies, linear search has linear time complexity, O(n), where n is length of the list.

Linear Search

- Linear search can be implemented using a simple for loop.

- Let's say, for example, that we want to implement our own version of Python's in keyword for checking membership in a list.

```
def is_in_list(query, items):
    for item in items:
        if item == query:
            return True

    return False
```

```
>>> is_in_list(4, [1, 4, 2, 3, 7])
True
>>> is_in_list(6, [1, 4, 2, 3, 7])

False
```

Linear Search

- Things get a little spicier when searching for an item based on a particular criterion (as opposed to an exact match).

- For example, let's say that we want to find a person by their full name.

Linear Search

```python
def find_by_name(people, name):
    for person in people:
        cur_name = f'{person.first_name} {person.last_name}'
        if cur_name == name:
            return person
    raise ValueError(f'person not found: {name}')
```

- Here we compute a value (cur_name) which is compared with the search key (name).
- When we find a matching person, we return the whole person object.
- If the person is not found we raise a ValueError.

Linear Search

```
people = [
    Person('Trent', 'Reznor', 55),
    Person('Atticus', 'Ross', 52),
    Person('Alessandro', 'Cortini', 44),
    Person('Chris', 'Vrenna', 53),
    Person('Charlie', 'Clouser', 57),
]
```

```
>>> chris = find_by_name(people, 'Chris Vrenna')
>>> print(str(chris))
Chris Vrenna, 53
>>> joe = find_by_name(people, 'Joe Blow')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in find_by_name
ValueError: 'person not found: Joe Blow'
```

Binary Search

- Although linear search is straightforward to implement, we can often come to a more efficient solution.
  - By this I mean that we can achieve a time complexity better than **O(n)**.

Binary Search

- Consider looking up a word in a dictionary.
- Do you start at page **1** and **go** through every word until you find the one you want?
  - I sure hope not!
  - But this is exactly what **linear search** does.

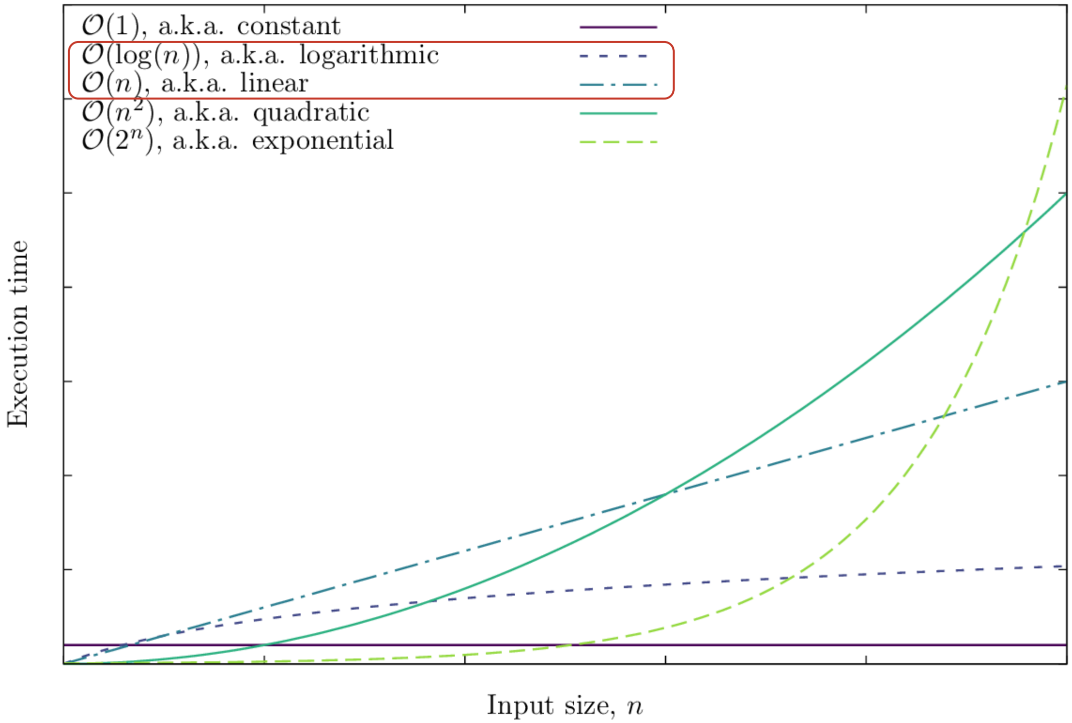
Binary Search

- Most people take advantage of the fact that words in a dictionary are ordered to speed up the search.

- We can take a similar approach to write a search algorithm which runs in O(log(n)) time ("logarithmic" time) when list items are already sorted.

- This is faster for large lists.

Binary Search vs. Linear Search Complexity



Linear search

Binary search

Implementing Binary Search

- Binary search halves the search space at each step.
- Start by considering the middle item in the list.
- If the item we are looking for comes after the middle item, we can disregard the entire first half of the list.
- This approach allows us to hone in on the item rapidly.

```python
def is_in_sorted_list(query, items):
    """Check whether `query` is
    in the sorted list `items`."""
    l = 0
    u = len(items)
    while u - l > 0:
        m = (u + l) // 2
        if items[m] == query:
            return True
        if query < items[m]:
            u = m
        else:
            l = m + 1

    return False
```

Implementing Binary Search

- Let's say that we are searching the list [1, 2, 3, 4, 7] for 4 (the "query").
- To begin with, the search region encompasses the entire list.

```
[1, 2, 3, 4, 7]
```

l=0 u=4

Implementing Binary Search

- We consider the middle item, which is halfway between l (the lower bound of the search region) and u (the upper bound of the search region).
  - m = (u + l) // 2

```
[1,  2,  3,  4,  7]
l=0      m=2     u=4
```

Implementing Binary Search

- Since the query is greater than the middle item, we move the lower bound of our search region up.

[1, 2, 3, 4, 7]

→ l=3 u=4

Implementing Binary Search

- The process repeats with the smaller search region.
- This time the middle item is **4**.

[1, 2, 3, 4, 7]

m=3 u=4

l=3

Implementing Binary Search

- Since our query now matches the middle item, we successfully found the item and can finish.

[1, 2, 3, 4, 7]

m=3 u=4

l=3

Implementing Binary Search

- Alternatively, if the query is not in the list, we will eventually reach an empty search region.
    - This corresponds to the case where the lower and upper bounds are the same (u - l == 0).

- In this case, we should stop searching and report that the query was not found.

Implementing Binary Search

```python
def is_in_sorted_list(query, items):
    """Check whether `query` is in the sorted list `items`."""
    l = 0                # Start lower bound at first list index.
    u = len(items)       # Start upper bound at last list index
    while u - l > 0: # While there is still list to search...
        m = (u + l) // 2  # Calculate the index of the middle item.
        if items[m] == query:  # If the middle item matches our query...
            return True        # ...return true.
        if query < items[m]:   # If our query is smaller than the middle item...
            u = m              # ...move the upper bound down.
        else:                  # Otherwise...
            l = m + 1          # ...move the lower bound up.
    return False     # Return false if we run out of list to search.
```

Implementing Binary Search

```
>>> is_in_sorted_list(4, [1, 2, 3, 4, 7])
True
>>> is_in_sorted_list(6, [1, 2, 3, 4, 7])
False
```

Limitations of Binary Search

- Warning: binary search is only guaranteed to find the item if the list being searched is **sorted correctly**.

```
>>> is_in_sorted_list(4, [1, 4, 2, 3, 7])
False
```

Recursion

Recursion

- Recursion is an approach to algorithm design which involves defining the solution to a problem in terms of a smaller instance of the same problem.
    - This can be solved by writing a function which calls itself.
- A function which calls itself is called a recursive function.
- The recursion must eventually stop when a solution is reached.

Base and Recursive Cases

- A base case is a branch of a recursive function which does not require further recursion.
  - Acts as an **end** to the recursion.

- A recursive case is a branch of a recursive function where the function must call itself one or more times.

Base and Recursive Cases

- Each step in the recursion must bring us closer to a base case in order for the recursion to eventually end.
  - If this doesn't happen, or there is no base case, the function will call **itself forever** (or, in practice, until the recursion limit is reached and the program **crashes**).

Example: Reversing a String

- Let's consider the problem of reversing a string.
- One solution is:
  - Take the first character and move it to the end of the string, and
  - Reverse the rest of the string.
- This is a recursive solution to the problem, because it involves solving a "smaller" instance of the same problem.

Example: Fibonacci Numbers

- Mathematically we define the nth Fibonacci number as follows:

$$F_n = F_{n-1} + F_{n-2} \text{ where } F_0 = 0 \text{ and } F_1 = 1$$

- This definition leads to a naturally recursive implementation in Python.

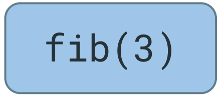
Example: Fibonacci Numbers

```
def fib(n):
    """Calculate F_n, the nth
    Fibonacci number.
    """
    ### Base cases.
    if n == 0:
        return 0
    if n == 1:
        return 1
    ### Recursive case.
    return fib(n - 1) + fib(n - 2)
```

- The base cases return without calling fib.
  - fib(0) returns 0.
  - fib(1) returns 1.

- The recursive case calls fib, but with a lower value of n (thus bringing us closer to a base case).
  - fib(n) returns fib(n - 1) + fib(n - 2).

Example: Fibonacci Numbers

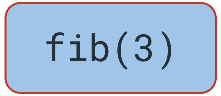


fib(3)

```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)
```

- Let's see what happens when Python executes fib(3).

Example: Fibonacci Numbers



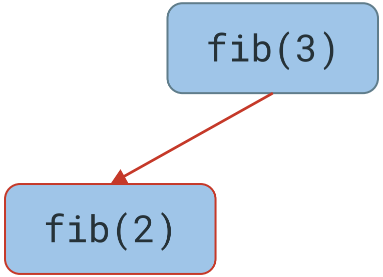
fib(3)

```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)
```

n = 3

- When n=3, we hit the recursive case.
- This involves two function calls, the first of which is fib(2) since n - 1= 2.

Example: Fibonacci Numbers

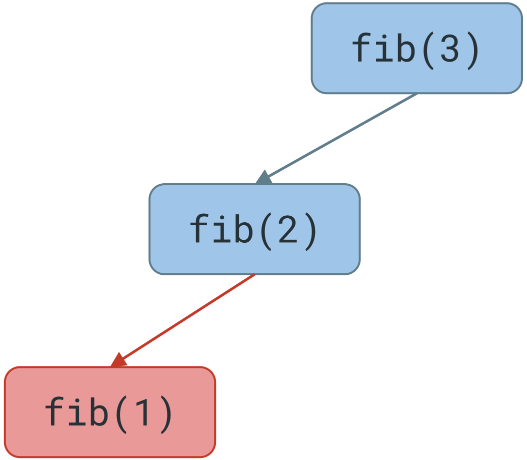


n = 2

```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)
```

- When n=2, we hit the recursive case again.
- This involves two function calls, the first of which is fib(1) since n - 1= 1.

Example: Fibonacci Numbers

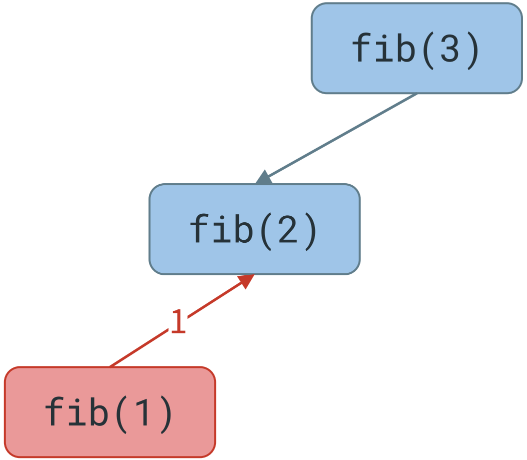


```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)
```

n = 1

- When n = 1, we hit the second base case.

Example: Fibonacci Numbers

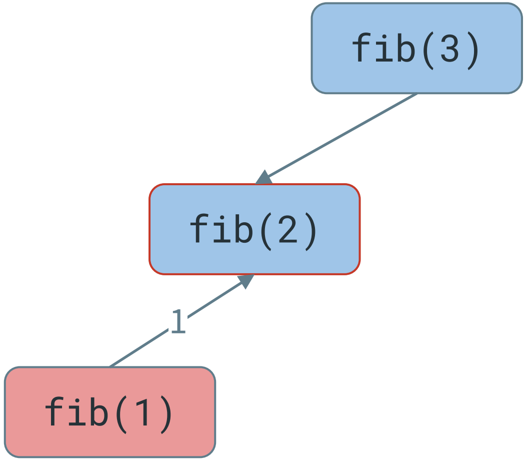


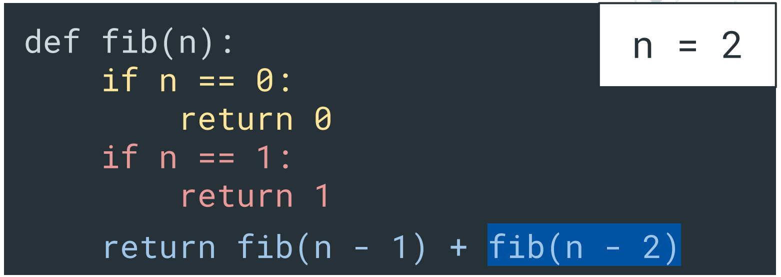
```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1

    return fib(n - 1) + fib(n - 2)
```

n = 1

- fib(1) returns 1.

Example: Fibonacci Numbers



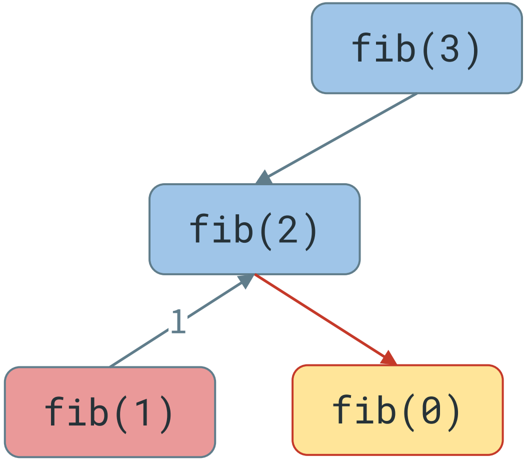


```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)
```

n = 2

- The second function call made by fib(2) is fib(0).

Example: Fibonacci Numbers

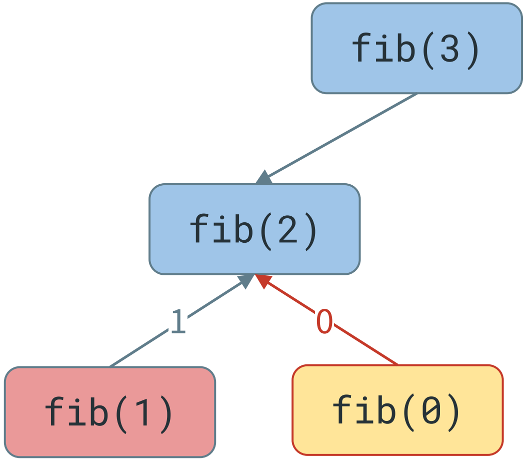


```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1

    return fib(n - 1) + fib(n - 2)
```

n = 0

- When n = 0, we hit the first base case.

Example: Fibonacci Numbers

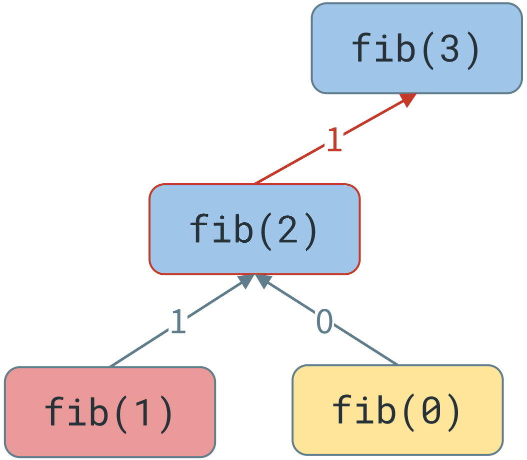


```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1

    return fib(n - 1) + fib(n - 2)
```

n = 0

- fib(0) returns 0.

Example: Fibonacci Numbers

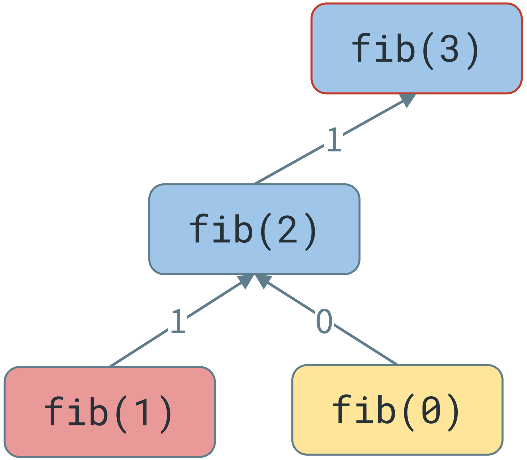


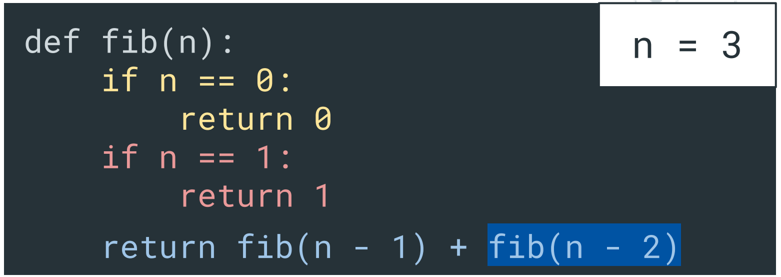
```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)
```

n = 2

- fib(2) returns 1, which is the sum of the two values returned from the recursive function calls.

Example: Fibonacci Numbers



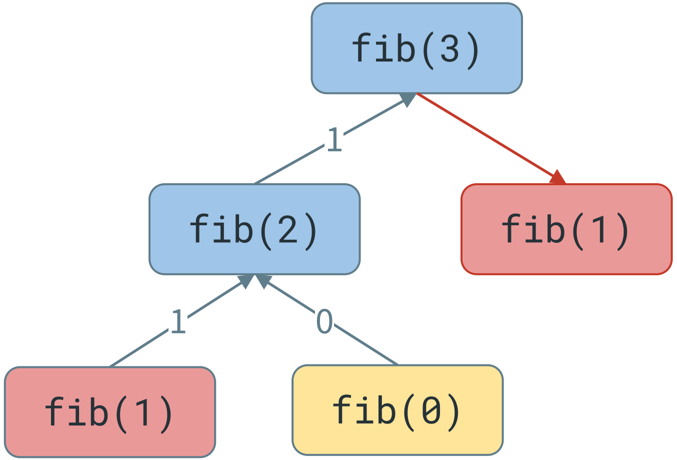


```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)
```

n = 3

- The second function call made by fib(3) is fib(1).

Example: Fibonacci Numbers

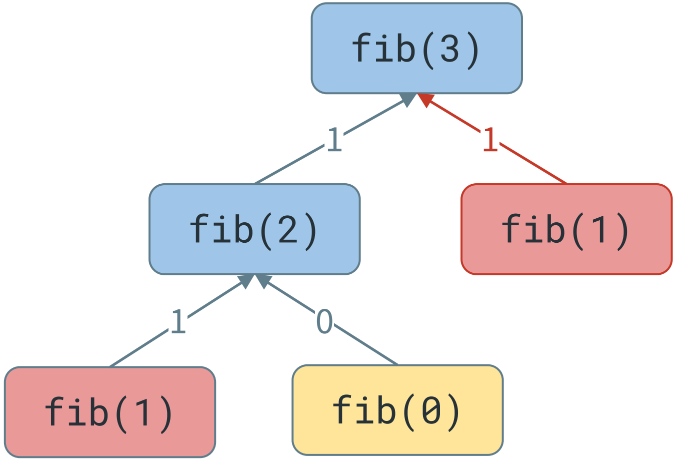


```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1

    return fib(n - 1) + fib(n - 2)
```

n = 1

- When n = 1, we hit the second base case.

Example: Fibonacci Numbers

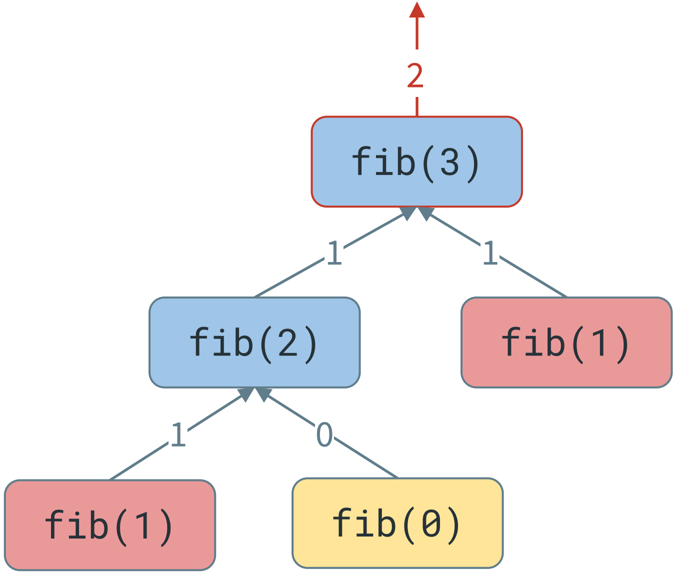


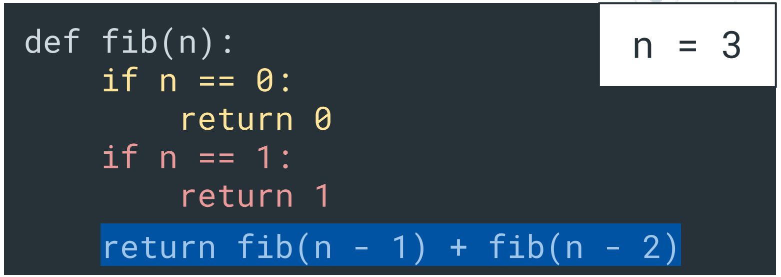
```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1

    return fib(n - 1) + fib(n - 2)
```

n = 1

- fib(1) returns 1.

Example: Fibonacci Numbers





```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)
```

n = 3

- fib(3) returns 2, which is the sum of the two values returned from the recursive function calls.

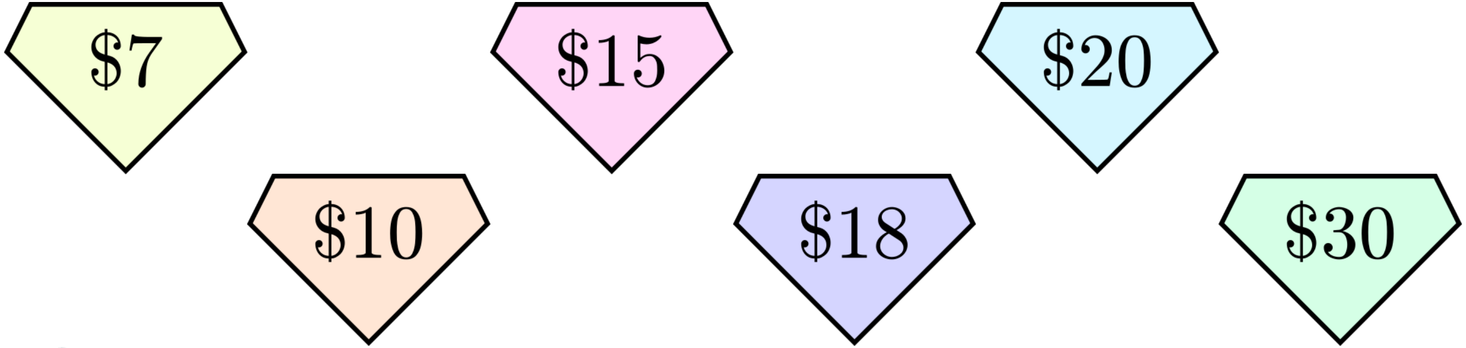
Recursive Solution to Binary Search

- We can use recursion to implement a binary search algorithm instead of the iterative solution presented earlier.
- The base cases are **1**) when the middle item matches the query, and **2**) when the search region is empty.
- Each step shrinks the search region, which brings us closer to a base case.

```python
def recursive_binary_search(query, items, l, u):
    # Base case 1: Search region is empty.
    if u - l <= 0:
        return False
    # Calculate middle index.
    m = (u + l) // 2
    # Base case 2: Found query.
    if items[m] == query:
        return True
    # Recursive case.
    if query < items[m]:
        # Move upper bound of search region down.
        u = m
    else:
        # Move lower bound of search region up.
        l = m + 1
    return recursive_binary_search(query, items, l, u)


def is_in_sorted_list(query, items):
    return recursive_binary_search(query, items, 0, len(items))
```

Dividing Loot (Redux)

- In the previous lecture we looked at the problem of dividing gems evenly between two people.



Dividing Loot (Redux)

Task description

Alice and Bob are famous explorers who frequently discover bags full of precious gemstones. Write a program which divides a bag of gemstones as evenly as possible between the two explorers, based on their values.

Dividing Loot (Redux)

- The greedy solution from last lecture was fast, but did not necessarily find the best way of dividing up gems.

- We will now look at a recursive solution which exhaustively considers all possible ways of dividing gems and picks the best one.

- Although this algorithm is slow, it is optimal and therefore guaranteed to find the best answer.

```python
def divide_gems_recursive(gems, gems_a=[], gems_b=[]):
    ### Base case.
    # If there are no gems to divide, simply return the gems
    # currently held by Alice and Bob.
    if len(gems) == 0:
        return gems_a, gems_b
    ### Recursive case.
    # Separate `gems` into the first item and the rest.
    first = gems[0]
    rest = gems[1:]
    # Option 1: Try giving the gem to Alice.
    gems_a1, gems_b1 = divide_gems_recursive(rest, gems_a + [first], gems_b)
    imbalance1 = abs(sum(gems_a1) - sum(gems_b1))
    # Option 2: Try giving the gem to Bob.
    gems_a2, gems_b2 = divide_gems_recursive(rest, gems_a, gems_b + [first])
    imbalance2 = abs(sum(gems_a2) - sum(gems_b2))
    # Pick whichever option led to a more balanced division.
    if imbalance1 <= imbalance2:
        return gems_a1, gems_b1
    else:

        return gems_a2, gems_b2
```

```
def divide_gems_recursive(gems, gems_a=[], gems_b=[]):
    ### Base case.
    # If there are no gems to divide, simply return the gems
    # currently held by Alice and Bob.
    if len(gems) == 0:
        return gems_a, gems_b
    ### Recursive case.
    # Separate `gems` into the
    first = gems[0]
    rest = gems[1:]
    # Option 1: Try giving the
    gems_a1, gems_b1 = divide_                                        _b)
    imbalance1 = abs(sum(gems_
    # Option 2: Try giving the
    gems_a2, gems_b2 = divide_                                        t])
    imbalance2 = abs(sum(gems_
    # Pick whichever option le
    if imbalance1 <= imbalance2:
        return gems_a1, gems_b1
    else:

        return gems_a2, gems_b2
```

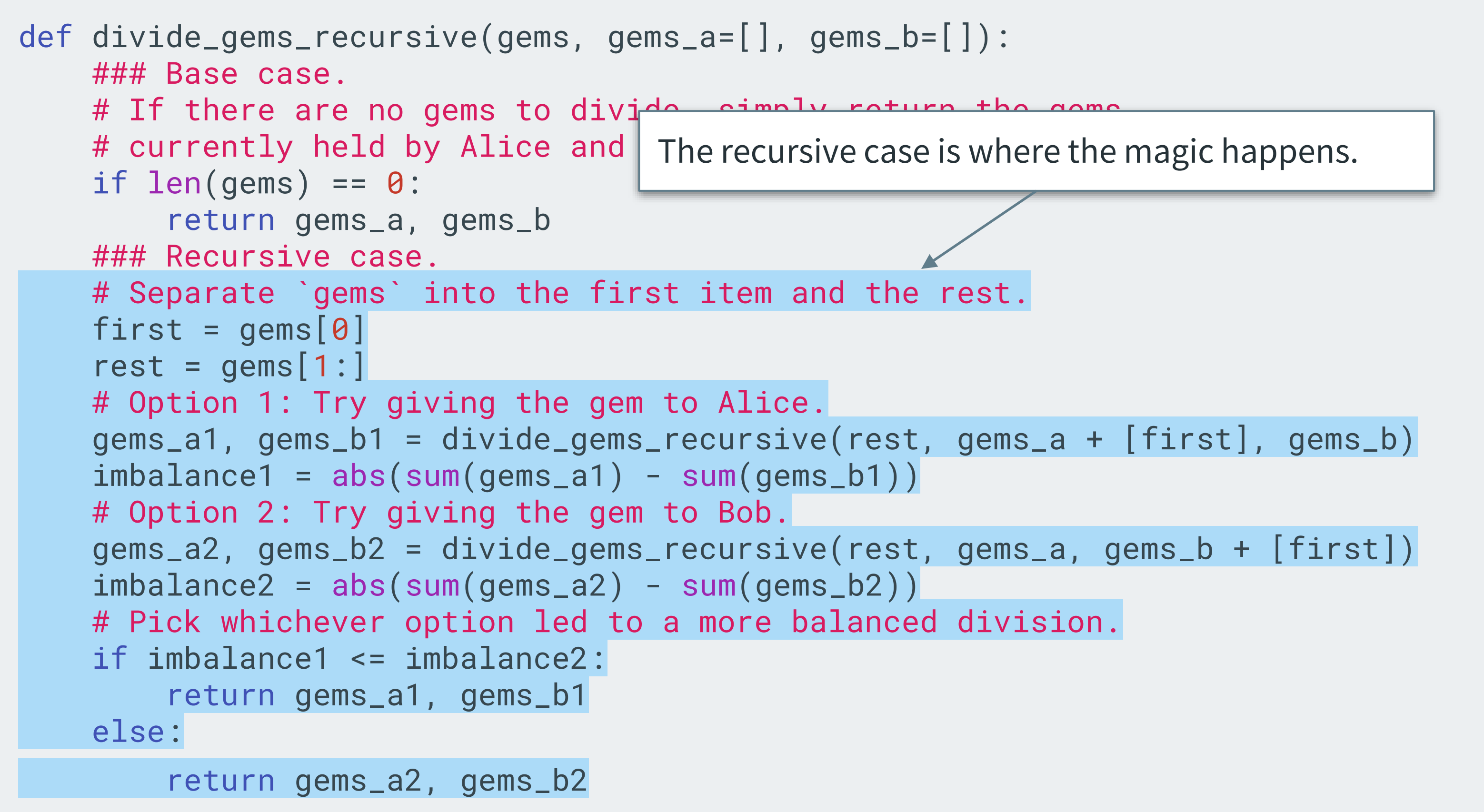
The function has parameters:

- `gems`, the remaining gems to divide.
- `gems_a`, the gems held by Alice (defaults to empty list).
- `gems_b`, the gems held by Bob (defaults to empty list).

The purpose of the function is to return the best way of dividing the remaining gems in `gems` between Alice and Bob given the gems that they already hold.

```
def divide_gems_recursive(gems, gems_a=[], gems_b=[]):
    ### Base case.
    # If there are no gems to divide, simply return the gems
    # currently held by Alice and Bob.
    if len(gems) == 0:
        return gems_a, gems_b
    ### Recursive case.
    # Separate `gems` into the first item and the rest.
    first = gems[0]
    rest = gems[1:]
    # Option 1:
    gems_a1, ge                                         + [first], gems_b)
    imbalance1
    # Option 2:
    gems_a2, gems_b2 = divide_gems_recursive(rest, gems_a, gems_b + [first])
    imbalance2 = abs(sum(gems_a2) - sum(gems_b2))
    # Pick whichever option led to a more balanced division.
    if imbalance1 <= imbalance2:
        return gems_a1, gems_b1
    else:
        return gems_a2, gems_b2
```

This is the base case where recursion ends. When we reach this, all gems have already been assigned to either Alice or Bob.



```
def divide_gems_recursive(gems, gems_a=[], gems_b=[]):
    ### Base case.
    # If there are no gems to divide, simply return the gems
    # currently held by Alice and
    if len(gems) == 0:
        return gems_a, gems_b
    ### Recursive case.
    # Separate `gems` into the first item and the rest.
    first = gems[0]
    rest = gems[1:]
    # Option 1: Try giving the gem to Alice.
    gems_a1, gems_b1 = divide_gems_recursive(rest, gems_a + [first], gems_b)
    imbalance1 = abs(sum(gems_a1) - sum(gems_b1))
    # Option 2: Try giving the gem to Bob.
    gems_a2, gems_b2 = divide_gems_recursive(rest, gems_a, gems_b + [first])
    imbalance2 = abs(sum(gems_a2) - sum(gems_b2))
    # Pick whichever option led to a more balanced division.
    if imbalance1 <= imbalance2:
        return gems_a1, gems_b1
    else:

        return gems_a2, gems_b2
```

The recursive case is where the magic happens.

```
def divide_gems_recursive(gems, gems_a=[], gems_b=[]):
    ### Base case.
    # If there                                              ems
    # currently
    if len(gems
        return gems_a, gems_b
    ### Recursive case.
    # Separate `gems` into the first item and the rest.
    first = gems[0]
    rest = gems[1:]
    # Option 1: Try giving the gem to Alice.
    gems_a1, gems_b1 = divide_gems_recursive(rest, gems_a + [first], gems_b)
    imbalance1 = abs(sum(gems_a1) - sum(gems_b1))
    # Option 2: Try giving the gem to Bob.
    gems_a2, gems_b2 = divide_gems_recursive(rest, gems_a, gems_b + [first])
    imbalance2 = abs(sum(gems_a2) - sum(gems_b2))
    # Pick whichever option led to a more balanced division.
    if imbalance1 <= imbalance2:
        return gems_a1, gems_b1
    else:
        return gems_a2, gems_b2
```

Firstly we separate the first gem from the rest of the list. The rest of the list could be empty.

Next we try giving the gem to Alice. This means that we add it to the gems that she holds, then perform a recursive call to divide the rest of the remaining gems.

```
def divide_gems_
    ### Base cas
    # If there a                                                   s
    # currently
    if len(gems)
        return gems_a, gems_b
    ### Recursive case.
    # Separate `gems` into the first item and the rest.
    first = gems[0]
    rest = gems[1:]
    # Option 1: Try giving the gem to Alice.
    gems_a1, gems_b1 = divide_gems_recursive(rest, gems_a + [first], gems_b)
    imbalance1 = abs(sum(gems_a1) - sum(gems_b1))
    # Option 2: Try giving the gem to Bob.
    gems_a2, gems_b2 = divide_gems_recursive(rest, gems_a, gems_b + [first])
    imbalance2 = abs(sum(gems_a2) - sum(gems_b2))
    # Pick whichever option led to a more balanced division.
    if imbalance1 <= imbalance2:
        return gems_a1, gems_b1
    else:
        return gems_a2, gems_b2
```

"What is the best way of dividing the rest of the gems if we give this one to Alice?"

```python
def divide_gems_recursive(gems, gems_a=[], gems_b=[]):
    ### Base case.
    # If there are no gems to divide, simply return the gems
    # currently
    if len(gems)
        return g
    ### Recursiv
    # Separate `gems` into the first item and the rest.
    first = gems[0]
    rest = gems[1:]
    # Option 1: Try giving the gem to Alice.
    gems_a1, gems_b1 = divide_gems_recursive(rest, gems_a + [first], gems_b)
    imbalance1 = abs(sum(gems_a1) - sum(gems_b1))
    # Option 2: Try giving the gem to Bob.
    gems_a2, gems_b2 = divide_gems_recursive(rest, gems_a, gems_b + [first])
    imbalance2 = abs(sum(gems_a2) - sum(gems_b2))
    # Pick whichever option led to a more balanced division.
    if imbalance1 <= imbalance2:
        return gems_a1, gems_b1
    else:

        return gems_a2, gems_b2
```

We calculate the imbalance for the "gem to Alice" hypothetical scenario.

```
def divide_gems_recursive(gems, gems_a=[], gems_b=[]):
    ### Base case.
    # If there are no gems to divide, simply return the gems
    # currently held by Alice and Bob.
    if len(gems) == 0:
        return
    ### Recursi
    # Separate
    first = gem
    rest = gems[1:]
    # Option 1: Try giving the gem to Alice.
    gems_a1, gems_b1 = divide_gems_recursive(rest, gems_a + [first], gems_b)
    imbalance1 = abs(sum(gems_a1) - sum(gems_b1))
    # Option 2: Try giving the gem to Bob.
    gems_a2, gems_b2 = divide_gems_recursive(rest, gems_a, gems_b + [first])
    imbalance2 = abs(sum(gems_a2) - sum(gems_b2))
    # Pick whichever option led to a more balanced division.
    if imbalance1 <= imbalance2:
        return gems_a1, gems_b1
    else:
        return gems_a2, gems_b2
```

We do the same thing, but for the scenario where we give the gem to Bob.

"What is the best way of dividing the rest of the gems if we give this one to Bob?"

```
def divide_gems_recursive(gems, gems_a=[], gems_b=[]):
    ### Base case.
    # If there are no gems to divide, simply return the gems
    # currently held by Alice and Bob.
    if len(gems) == 0:
        return gems_a, gems_b
    ### Recursive case.
    #                                              e rest.
    f
    r
    #
    g                                             t, gems_a + [first], gems_b)
    imbalance1 = abs(sum(gems_a1) - sum(gems_b1))
    # Option 2: Try giving the gem to Bob.
    gems_a2, gems_b2 = divide_gems_recursive(rest, gems_a, gems_b + [first])
    imbalance2 = abs(sum(gems_a2) - sum(gems_b2))
    # Pick whichever option led to a more balanced division.
    if imbalance1 <= imbalance2:
        return gems_a1, gems_b1
    else:
        return gems_a2, gems_b2
```

Lastly, we compare the outcomes of the two scenarios and return the one which resulted in better balance.

Dividing Loot (Redux)

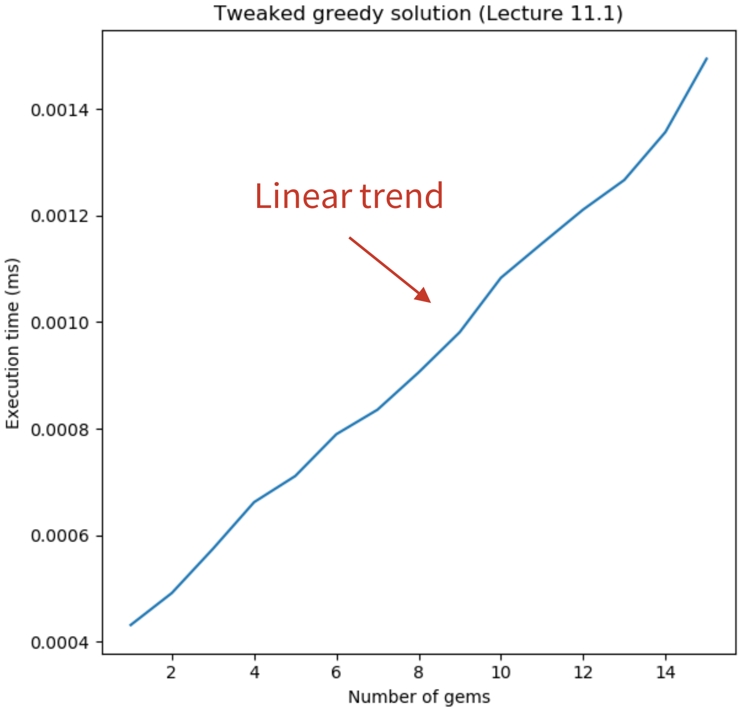
```
>>> gems_a, gems_b = divide_gems_recursive([30, 20, 18, 15, 10, 7])
>>> gems_a
[30, 20]
>>> gems_b
[18, 15, 10, 7]
>>> sum(gems_a)
50
>>> sum(gems_b)
50
```

- As expected, our new solution gives the best possible answer (in this case, perfect balance).
- However, this solution is slow for large lists.

Algorithm Efficiency Comparison

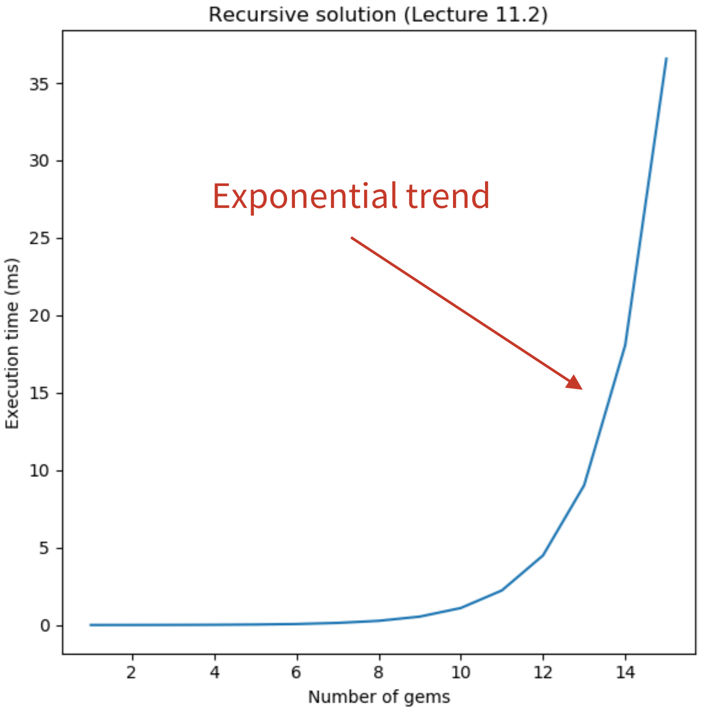
- To demonstrate the difference in speed, I conducted a little experiment.
- I ran both the recursive and greedy solutions on different numbers of gems and timed them.
  - The greedy solution incorporates the improvement from last lecture's Check Your Understanding question.
- The plot shows...

Algorithm Efficiency Comparison



Tweaked greedy solution (Lecture 11.1)

Linear trend



Recursive solution (Lecture 11.2)

Exponential trend

Algorithm Efficiency Comparison

- For a list of 25 gems, the recursive solution takes a patience-testing **38 seconds**!

- On the other hand, the greedy solution takes an instantaneous **4 microseconds**.

- So, in this case, **optimality** comes at a **high** performance **cost**.

Recursion Limit

- Recall from an earlier lecture that Python must keep a call stack so that it knows where to return to once a function call finishes.
- Recursive function calls can cause the call stack to grow very large.
- Python imposes a recursion limit.
  - By default this is ~1000.
- Reaching the limit will result in a `RecursionError`.

Summary of Recursion

- A recursive function must include:
  - At least one base case, and
  - At least one recursive case.

- The recursive case(s) must in some way bring us closer to a base case.
  - If not, the recursion will not finish (in practice this means crashing with a RecursionError).

Next Lecture We Will…

- No Next Lecture ☺

Exam Date and Instructions

- Exam date: 19/04/2023 – 5:50 PM to 8:15 PM (Melb time)

- Exam instructions: please check LMS announcements and the attached file.

Thanks for your attention!

The slides and lecture recording will be made available on LMS.

PPT Acknowledgement: Dr Aiden Nibali, CS&IT LTU.