# Algorithms

CMPT 145

# Algorithms

- An algorithm is a sequence of instructions that accomplish a stated task.
- Example tasks:
  - Calculate the average of a collection of numbers
  - Calculate the square root of a number
  - Check if a binary tree is ordered.

How do you design an algorithm
if you do not already know
how the algorithm should work?

Study algorithms designed by someone else.

# Algorithms Unit Overview

1. Tasks: What kinds of tasks do we write algorithms for?
2. Algorithm Styles: What kinds of algorithms are there?
3. Examples: We study example algorithms for a variety of tasks.

# Lecture Overview

- Subset Sum
- Maximum Slice
- Making Change
- Maximum Tree Path
- Leap Line

# Subset Sum

- Given:
  - List of positive numbers, $L$
  - Target value $T$
- Find a list of numbers $M$, taken from $L$, whose sum is exactly $T$.

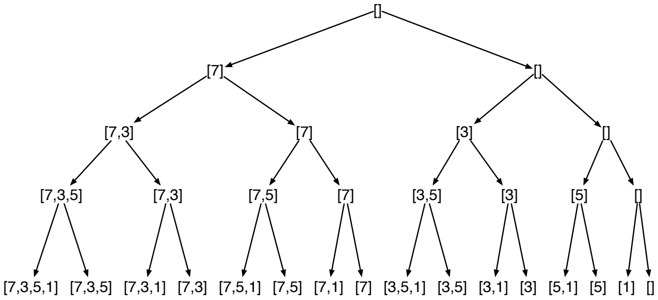
- Example:
  - $L = [1, 3, 5, 7]$
  - $T = 8$
- Solution: $M = [1, 7]$.

- The solution, $M$, is a list, and we can construct a solution by inserting numbers from $L$.

# Brute Force

- Try every possible subset.
- Return one that sums to target.

# Every Possible Subset




A tree of all the possible subsets of the list $[1, 3, 5, 7]$. Each left branch includes one of the elements; each right branch leaves it out. The number of levels is $N + 1$. The number of leaf nodes is $2^N$.

# Every Possible Subset

```
def all_subsets(alist):
    """
    Purpose:
        Given a list, display all subsets.
    Preconditions:
        alist: a list
    Post-conditions:
        displays all subsets to the console,
        one subset per line.  Could be a lot!
    Return:
        None
    """
```

# Every Possible Subset

```python
def all_subsets(alist):
    def allsub(al, sub):
        if len(al) == 0:
            print(sub)
        else:
            first = al[0]
            rest = al[1:]
            allsub(rest, [first]+sub)  # with first
            allsub(rest, sub)          # without first

    allsub(alist,[])
```

A pre-order traversal! The tree is not stored in memory; it's conceptual.

# Brute Force Algorithm: Subset Sum

```
def subsetsum_v1(alist, target):
    """
    Purpose:
        Given a list of positive integers, and a target,
        Return a sublist of integers whose sum is target.
    Preconditions:
        alist: a list of positive integers
        target: a positive integer
    Return:
        A tuple True, sublist if sum(sublist) == target
        Or False, None otherwise
    """
```

# Brute Force

```python
def subsetsum_v1(alist, target):

    def trysum(al, subset):
        if len(al) == 0 and sum(subset) == target:
            return True, subset
        elif len(al) == 0:
            return False, None
        else:
            first = al[0]
            rest = al[1:]
            flag, answer = trysum(rest, subset + [first])
            if flag:
                return flag, answer
            else:
                # try without the first value
                return trysum(rest, subset)

    return trysum(alist, [])
```

# Notes on Brute Force version

- Try every possible subset.
  - There are $2^N$ subsets of $N$ values.
  - In the worst case, we have to look at them all.
  - Worst case time complexity: $O(2^N)$
  - This is really really bad.
- Many subsets may have sums much too large. Why not stop early?

# Backtracking

- Try every possible subset.
- Return one that sums to target.
- If a subset's sum is too large, try another.

# Backtracking

```python
def subsetsum_v2(alist, target):

    def trysum(al, subset):
        if sum(subset) == target:
            return True, subset
        elif len(al) == 0 or sum(subset) > target:
            return False, None
        else:
            first = al[0]
            rest = al[1:]
            flag, answer = trysum(rest, subset + [first])
            if flag:
                return flag, answer
            else:
                # try without the first value
                return trysum(rest, subset)

    return trysum(alist, [])
```

# Subset Sum output: Typical

```
Brute Force (Version 1) on list of size 22 :
Target: 11695
Time: 2.17
Result: (True, [3597, 3001, 3600, 1497])

Backtracking (Version 2) on list of size 22 :
Target: 11695
Time: 0.0148
Result: (True, [3597, 3001, 3600, 1497])
```

# Subset Sum output: Easy

```
Brute Force (Version 1) on list of size 22 :
Target: 25422
Time: 0.0588
Result: (True, [3597, 2684, 3622, 124, 2936, 3001, 2317,
2304, 994, 1094, 2749])

Backtracking (Version 2) on list of size 22 :
Target: 25422
Time: 0.00343
Result: (True, [3597, 2684, 3622, 124, 2936, 3001, 2317,
2304, 994, 1094, 2749])
```

# Subset Sum output: No solution, small target

```

Brute Force (Version 1) on list of size 22 :
Target: 7137
Time: 4.621402
Result: (False, None)

Backtracking (Version 2) on list of size 22 :
Target: 7137
Time: 0.00726900000000858
Result: (False, None)
```

# Subset Sum output: No solution large target

```

Brute Force (Version 1) on list of size 22 :
Target: 259420
Time: 4.633568999999998
Result: (False, None)

Backtracking (Version 2) on list of size 22 :
Target: 259420
Time: 5.813278
Result: (False, None)
```

# Notes on Backtracking version

- Try every possible subset.
  - There are $2^N$ subsets of $N$ values.
  - In the worst case, we have to look at them all.
  - Worst case time complexity: $O(2^N)$
  - This is really really bad.
- On average, this version is better, if:
  - There is a solution or,
  - Most subsets have sums larger than the target.
- But the worst case is no better than Brute Force.

Backtracking can be effective, but no guarantees.

# Maximum Slice

- Given a list of numbers, $L$
- Find the slice from index $a$ to index $b$ that has the largest sum of all possible slices of $L$.

- Example: $L = [1, -2, 3, 4, -5]$
- Solution: $L[2:4]$

- The solution $L[a:b]$ can be constructed by exploring different indices $a$ and $b$.

# Brute Force

- Try every possible slice.
- Return the slice with highest sum.

# Every Possible Slice

```python
def allslices(alist):
    """
    Purpose:
        Display all slices to the console.
    Preconditions:
        alist: a list of numbers
    Post-conditions:
        Outputs all slices to the console.
    Return:
        None
    """
    for a in range(len(alist)):
        for b in range(a,len(alist)):
            print(alist[a:b+1])
```

# Brute Force, version 0

```python
def maxslice_brute_force_v0(alist):
    """
    Purpose:
        Find the maximum sum of all slices of alist.
    Preconditions:
        alist: a list of numbers
    Post-conditions:
        None
    Return:
        a number, the maximum slice sum
    """
    maxsum = alist[0]
    for i in range(len(alist)):
        for j in range(i+1, len(alist)):
            slice = sum(alist[i:j + 1])
            if slice > maxsum:
                maxsum = slice
    return maxsum
```

# Brute Force, version 0

- Try every possible slice.
  - Let $N$ be the length of the input list.
  - A slice can start at any index $i$ ($N$ starts)
  - A slice can end at any index after $i$ ($N - i$)
  - There are $O(N^2)$ slices!
  - In the worst case, calling `sum(alist[i:j+1])` is $O(N)$.
  - Version 0: worst-case time complexity: $O(N^3)$.

# Brute Force, version 1

- Try every possible slice.
- Return the slice with highest sum.
- Be smarter about calculating sums.

Which script requires more steps?

```
1 # script 1
2 alist = [1, 2, 3, 4, 5]
3 sum3 = sum(alist[0:3])
4 sum4 = sum3 + alist[4]
```

```
1 # script 2
2 alist = [1, 2, 3, 4, 5]
3 sum3 = sum(alist[0:3])
4 sum4 = sum(alist[0:4])
```

Script 2 Line 4 repeats all the additions done on Line 3!

# Brute Force, version 1

- Strategy: Store partial sums in a dictionary
- A new dictionary for each starting slice $i$
- Key: Index $j$ last index for a slice
- Value: the sum for the slice $i, j$.
- Benefit: avoid $O(N)$ cost of `sum()`
- Cost: $O(N)$ memory

# Brute Force, version 1

```python
def maxslice_brute_force_v1(alist):
    """
    Purpose:
        Find the maximum sum of all slices of alist.
    Preconditions:
        alist: a list of numbers
    Post-conditions:
        None
    Return:
        a number, the maximum slice sum
    """
```

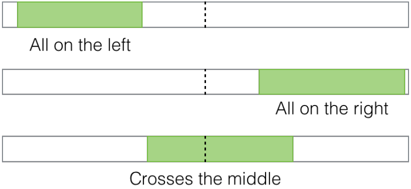
# Brute Force, version 1

```
def maxslice_brute_force_v1(alist):
    # using brute force: look at all possible slices
    # but store all the partial sums in a dictionary
    # where s[j] stores the value sum(alist[i,j+1])

    maxsum = alist[0]
    for i in range(len(alist)):
        s = {}
        s[i] = alist[i]
        if s[i] > maxsum:
            maxsum = s[i]
        for j in range(i+1, len(alist)):
            s[ j] = s[j-1] + alist[j]
            if s[j] > maxsum:
                maxsum = s[j]
    return maxsum
```

# Maximum Slice output: Typical

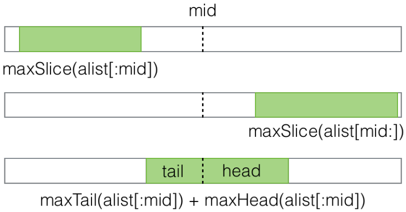
```
Example: list of length: 1000
Brute Force version 0:
Result: 1015 Time: 1.92

Brute Force version 1:
Result: 1015 Time: 0.364

Example: list of length: 2000
Brute Force version 0:
Result: 1913 Time: 15.4   # 2x length, 8x time

Brute Force version 1:
Result: 1913 Time: 1.45   # 2x length, 4x time
```

# Divide and conquer: Maximum Slice




- Key idea: Split the list into 2 roughly equal halves.
- The maximum slice can be found in three places:
    1. All on the left
    2. All on the right
    3. Crossing the middle

# Divide and conquer: Maximum Slice




- The maximum slice can be found in three places.
- We don't know where it is, so try all three:

  1. Recursively, on left and right halves.
  2. `maxTail()`: Find the best slice ending at `mid`.
  3. `maxHead()`: Find the best slice starting at `mid`.

# Divide and Conquer

```python
def maxslice_DC(alist):
    """
    Purpose:
        Find the maximum sum of all slices of alist.
    Preconditions:
        alist: a list of numbers
    Post-conditions:
        None
    Return:
        a number, the maximum slice sum
    """
```

# Divide and Conquer

```python
def maxslice_DC(alist):
    # internal function
    def max_tail(left, right):
        """
        Calculate the maximum slice that ends at right
        (from any point starting at left or later)
        """
        s = {}
        s[right] = alist[right]
        maxsum = s[right]
        # calculate the sums from right to left (backwards)
        for i in range(right - 1, left - 1, -1):
            s[i] = s[i + 1] + alist[i]
            if s[i] > maxsum:
                maxsum = s[i]
        return maxsum
```

# Divide and Conquer

```python
def maxslice_DC(alist):
    # internal function
    def max_head(left, right):
        """
        Calculate the maximum slice that starts at left
        (to any point up to and including right)
        """
        s = {}
        s[left] = alist[left]
        maxsum = s[left]
        for i in range(left + 1, right + 1):
            s[i] = s[i - 1] + alist[i]
            if s[i] > maxsum:
                maxsum = s[i]
        return maxsum
```

# Divide and Conquer

```
def maxslice_DC(alist):
    # internal function
    def maxslice_rec(left, right):
        """
        Recursively find maximum slice between left and right.
        """
        # using divide and conquer
        if left == right:
            return alist[left]
        else:
            # divide, and solve
            mid = (right + left) // 2
            max_left  = maxslice_rec(left, mid)
            max_right = maxslice_rec(mid + 1, right)
            max_cross = (max_tail(left, mid)
                        + max_head(mid + 1, right))
            # conquer
            return max(max_left, max_right, max_cross)
```

# Divide and Conquer

```python
def maxslice_DC(alist):
    # body of maxslice_DC
    return maxslice_rec(0, len(alist) - 1)
```

# Maximum Slice output: Typical

```
Example: list of length: 1000
Brute Force version 0:
Result: 1795 Time: 1.90

Brute Force version 1:
Result: 1795 Time: 0.103

Divide and conquer:
Result: 1795 Time: 0.00360

Example: list of length: 2000
Brute Force version 0:
Result: 4994 Time: 15.4    # 2x length, 8x time

Brute Force version 1:
Result: 4994 Time: 0.402   # 2x length, 4x time

Divide and conquer:
Result: 4994 Time: 0.00772 # 2x length, 2x time
```

# Divide and conquer: Maximum Slice

- Timing evidence suggests time complexity of $O(N)$.
- Formal analysis proves $O(N)$
- Beyond first year expectations!

# Making Change

- Given:
  - Positive integer $D$
  - A list $L$ of coin values
- Find a list of integers $C$, indicating how many of each coin are needed to have the value of $D$ exactly.

- Example:
  - $D = 37$
  - $L = [1, 5, 10, 25]$
- Solution: $C = [2, 2, 0, 1]$.

# Making Change: A Natural, Greedy Algorithm

- Brute force: try all combinations.
  - Possible, but obviously weak.
- Greedy algorithm:
  - Use as many large coins as possible.
  - Try smaller coins on remaining amounts.
- It's greedy because it commits to a choice now, made without considering future choices.

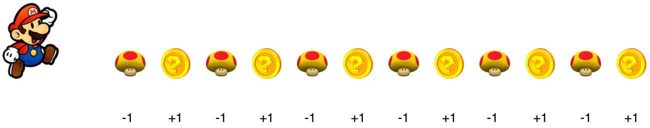
# Making Change: Greedy

```python
def change_v2(cents):
    """
    Purpose:
        Make change for the given cents value.
        Assumes coin values 25c, 10c, 5c, 1c
    Pre-conditions:
        :param cents: an integer
    Return:
        a list of counts for the coins used.
    """
    coins = [25, 10, 5, 1]
    coin_index = 0
    counts = [0] * len(coins)
    remaining = cents
    while remaining > 0:
        counts[coin_index] = remaining // coins[coin_index]
        remaining = remaining % coins[coin_index]
        coin_index += 1
    return counts
```

# Making Change: Time complexity

- Exercise. What is the time complexity of `change_v2()`?

# Leap Line




- Given: a sequence of coins/mushrooms (positive and negative numbers).
- Mario can step on, or jump over, an item.
- What's Mario's highest point total?

# Leap line: Brute Force

- Try every possible sequence of steps and jumps.
- Return the sequence with highest score.
- Exercise: How many possible sequences are there?

# Leap line: Brute Force

- Try every possible sequence of steps and jumps.
- Brute force algorithm:
    1. Collect the points at `loc`
    2. Recursively, find the best score possible by stepping from `loc`
    3. Recursively, find the best score possible by jumping from `loc`
    4. Pick whichever is bigger.

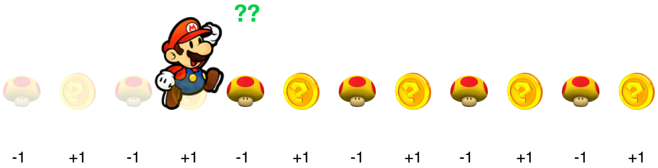
# Leap line: Brute Force

```python
def maximumScoreFrom_v0(track):
    """
    Purpose:
        Calculate the maximum score that can be obtained from
        stepping and jumping along the given track
    Pre-conditions:
        :param track: a list of integers
    Return:
        the maximum score
    """
```

# Leap line: Brute Force

```python
def maximumScoreFrom_v0(track):
    def jump_or_step(loc):
        """
        Calculate the maximum score that can be obtained from
        starting at the given location
        """
        step_loc = loc + 1
        jump_loc = loc + 2

        if step_loc == len(track):
            return track[loc]
        elif jump_loc >= len(track):
            return track[loc] + jump_or_step(step_loc)
        else:
            return track[loc] + max(jump_or_step(step_loc),
                                    jump_or_step(jump_loc))

    return jump_or_step(0)
```

# Leap line: Storing your work for future use




Mario should calculate the best sequence of choices from here to the end once, and then save it. Every other time he explores this sub-problem, he can look up the answer he saved. This is called Dynamic programming.

# Leap line: Dynamic Programming

```
def maximumScoreFrom_v1(track):
    # body of maximumScoreFrom_v1()
    # using memoization
    # memo[loc] stores the best score starting from loc.
    memo = {}
    return jump_or_step(0)
```

# Leap line: Dynamic Programming

```python
def maximumScoreFrom_v1(track):
    def jump_or_step(loc):
        # check if the best score is already known
        if loc in memo:
            return memo[loc]

        step_loc = loc + 1
        jump_loc = loc + 2

        if step_loc == len(track):
            return track[loc]
        elif jump_loc >= len(track):
            result = track[loc] + jump_or_step(step_loc)
            memo[loc] = result
            return result
        else:
            result = track[loc] + max(jump_or_step(step_loc),
                                      jump_or_step(jump_loc))
            memo[loc] = result
            return result
```

# Leap Line output: Typical

```
Example: list of length: 14
Brute Force version (v0):
Result: 1
Time: 0.0003

Dynamic programming (v1):
Result: 1
Time: 1.6e-05

Example: list of length: 37
Brute Force version (v0):
Result: 8
Time: 19.3          # 2.6x length 64000x time

Dynamic programming (v1):
Result: 8
Time: 5.7e-05       # 2.6x length 3.5x time
```

# Dynamic Programming: Leap Line

- There are $O(2^N)$ sequences; much repeated work!
- Brute force algorithm looks at all of them!
- Dynamic programming:
  - Calculates the best sequence from `loc` once.
  - There are $N$ locations.
  - Time complexity: $O(N)$.
  - Cost: the dictionary stores $O(N)$ best scores.

Dynamic programming = brute force + memoization.

# Summary – Algorithm Styles

- Brute force: try all possible choices.
- Backtracking: prevent bad choices.
- Greedy: commit to a choice that seems good.
- Divide and Conquer: split your problem in half.
- Dynamic programming: Brute force + memoization

# Summary: How to solve it

- Try Brute force as prototype. You can learn a lot.
- Brute force too slow? Try Backtracking.
- Backtracking too slow? Try Greedy.
- Greedy gives poor solution? Try Divide and Conquer.
- Divide and Conquer doesn't help? Try Dynamic programming.
- If all these fail, it might be a hard problem!