COMP110: Principles of Computing

10: Algorithm Strategies

Worksheets

- Worksheet 6: due **this Friday**
- Worksheet 7: due **next Friday**

Recursion

Recursion

Recursion

- A **recursive** function is a function that **calls itself**

Recursion

- A **recursive** function is a function that **calls itself**
- Example: the **Fibonacci numbers** — each number in the sequence is the sum of the previous two

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

Recursion

- A **recursive** function is a function that **calls itself**
- Example: the **Fibonacci numbers** — each number in the sequence is the sum of the previous two

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

- To calculate the *n*th Fibonacci number:

```
int fibonacci(int n)
{
    if (n <= 2)
        return 1;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}
```

Recursion

- A **recursive** function is a function that **calls itself**
- Example: the **Fibonacci numbers** — each number in the sequence is the sum of the previous two

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

- To calculate the *n*th Fibonacci number:

```
int fibonacci(int n)
{
    if (n <= 2)
        return 1;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}
```

- Recursive functions need a **base case** where they stop recursing, otherwise they will go **forever**

Thinking recursively

Thinking recursively

- I want to solve a problem

Thinking recursively

- I want to solve a problem
- If I already had a function to solve smaller instances of the problem, I could use it to write my function

Thinking recursively

- I want to solve a problem
- If I already had a function to solve smaller instances of the problem, I could use it to write my function
- I can solve the smallest possible problem

Thinking recursively

- I want to solve a problem
- If I already had a function to solve smaller instances of the problem, I could use it to write my function
- I can solve the smallest possible problem
- Therefore I can write a recursive function

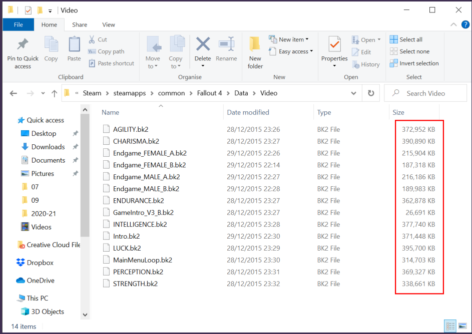
Example: file sizes

Example: file sizes

- Suppose we want to find the total size of all files in a folder and its subfolders

Example: file sizes

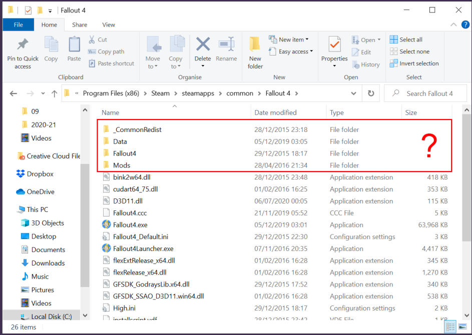
- Suppose we want to find the total size of all files in a folder and its subfolders



- If the folder contains **only** files, then we can simply add their sizes together

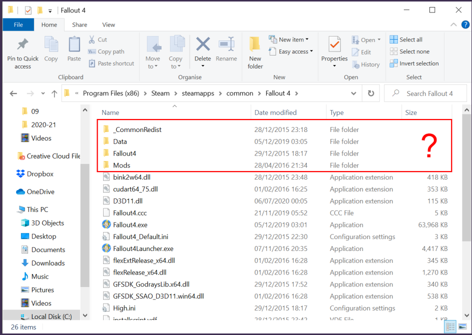
Example: file sizes

- What if the folder contains subfolders?



Example: file sizes

- What if the folder contains subfolders?



- We need to find the total size of all files in the subfolders and their subsubfolders...

Example: file sizes — recursive solution

assume the system provides a GETFILESIZE function

Example: file sizes — recursive solution

assume the system provides a GETFILESIZE function

procedure CALCULATEFOLDERSIZE(*folder*)

Example: file sizes — recursive solution

assume the system provides a GETFILESIZE function
procedure CALCULATEFOLDERSIZE(*folder*)
    totalSize $\leftarrow$ 0

Example: file sizes — recursive solution

assume the system provides a GETFILESIZE function

procedure CALCULATEFOLDERSIZE(*folder*)

    totalSize ← 0

    for each *item* in *folder* **do**

Example: file sizes — recursive solution

assume the system provides a GETFILESIZE function

procedure CALCULATEFOLDERSIZE(*folder*)
    totalSize ← 0
    for each *item* in *folder* **do**
        if *item* is a file **then**
            totalSize ← *totalSize* + GETFILESIZE(*item*)

Example: file sizes — recursive solution

assume the system provides a GETFILESIZE function

```
procedure CALCULATEFOLDERSIZE(folder)
    totalSize ← 0
    for each item in folder do
        if item is a file then
            totalSize ← totalSize + GETFILESIZE(item)
        else if item is a folder then
            totalSize ← totalSize + CALCULATEFOLDERSIZE(item)
```

Example: file sizes — recursive solution

assume the system provides a GETFILESIZE function

```
procedure CALCULATEFOLDERSIZE(folder)
    totalSize ← 0
    for each item in folder do
        if item is a file then
            totalSize ← totalSize + GETFILESIZE(item)
        else if item is a folder then
            totalSize ← totalSize + CALCULATEFOLDERSIZE(item)
        end if
    end for
    return totalSize
```

Example: file sizes — recursive solution

assume the system provides a GETFILESIZE function
procedure CALCULATEFOLDERSIZE(*folder*)
    totalSize ← 0
    for each *item* in *folder* **do**
        if *item* is a file **then**
            totalSize ← *totalSize* + GETFILESIZE(*item*)
        else if *item* is a folder **then**
            totalSize ← *totalSize* + CALCULATEFOLDERSIZE(*item*)
        end if
    end for
    return totalSize
end procedure

The call stack

The call stack

- Recall: nested function calls are handled using a **stack**

The call stack

- Recall: nested function calls are handled using a **stack**
- **Calling** a function **pushes** a frame onto the stack

The call stack

- Recall: nested function calls are handled using a **stack**
- **Calling** a function **pushes** a frame onto the stack
- **Returning** from a function **pops** the top frame from the stack

The call stack

- Recall: nested function calls are handled using a **stack**
- **Calling** a function **pushes** a frame onto the stack
- **Returning** from a function **pops** the top frame from the stack
- Recursive functions are no different

The call stack

- Recall: nested function calls are handled using a **stack**
- **Calling** a function **pushes** a frame onto the stack
- **Returning** from a function **pops** the top frame from the stack
- Recursive functions are no different
- This means if a recursive function contains **local variables**, they are **independent** between instances of the function

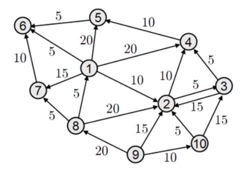
The call stack

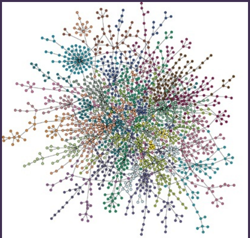
- Recall: nested function calls are handled using a **stack**
- **Calling** a function **pushes** a frame onto the stack
- **Returning** from a function **pops** the top frame from the stack
- Recursive functions are no different
- This means if a recursive function contains **local variables**, they are **independent** between instances of the function
- This is also why careless use of recursion can lead to a **stack overflow**

Graphs and trees

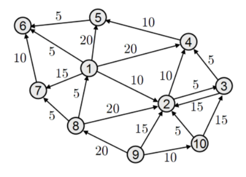
Graphs

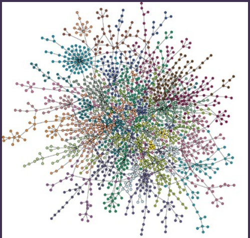
Graphs





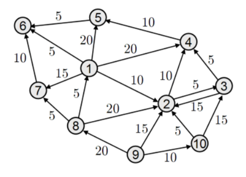
Graphs

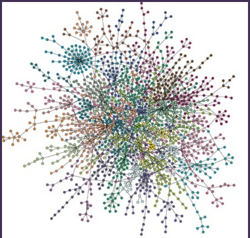




- A **graph** is defined by:

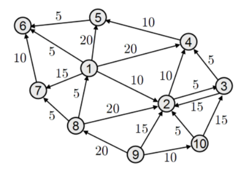
Graphs

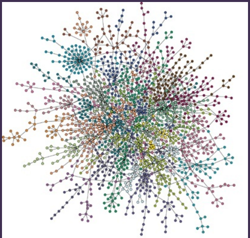




- A **graph** is defined by:
  - A collection of **nodes** or **vertices** (points)

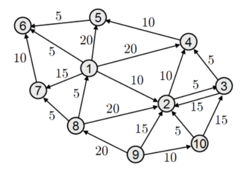
Graphs

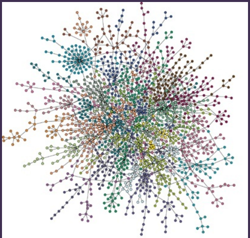




- A **graph** is defined by:
  - A collection of **nodes** or **vertices** (points)
  - A collection of **edges** or **arcs** (lines or arrows between points)

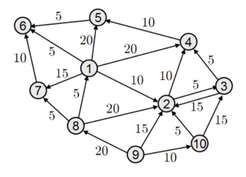
Graphs

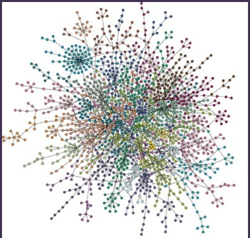




- A **graph** is defined by:
  - A collection of **nodes** or **vertices** (points)
  - A collection of **edges** or **arcs** (lines or arrows between points)
- Often used to model **networks** (e.g. social networks, transport networks, game levels, automata, ...)

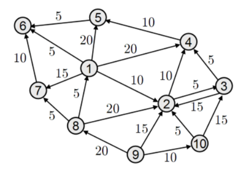
Graphs

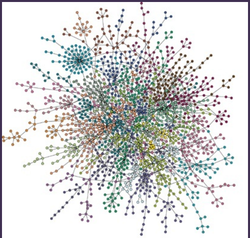




- A **graph** is defined by:
  - A collection of **nodes** or **vertices** (points)
  - A collection of **edges** or **arcs** (lines or arrows between points)
- Often used to model **networks** (e.g. social networks, transport networks, game levels, automata, ...)
- **Directed** graph: edges are arrows

Graphs





- A **graph** is defined by:
  - A collection of **nodes** or **vertices** (points)
  - A collection of **edges** or **arcs** (lines or arrows between points)
- Often used to model **networks** (e.g. social networks, transport networks, game levels, automata, ...)
- **Directed** graph: edges are arrows
- **Undirected** graph: edges are lines

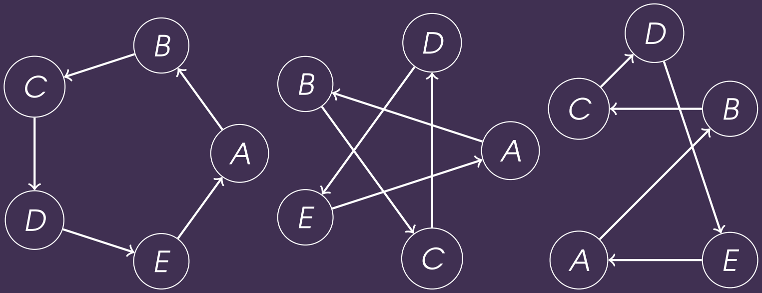
Drawing graphs

Drawing graphs

- A graph does not necessarily specify the physical **positions** of its nodes

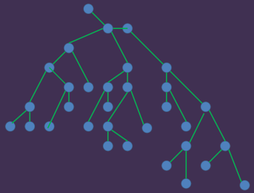
Drawing graphs

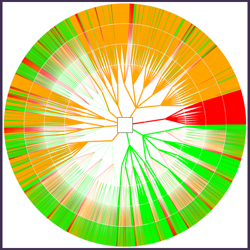
- A graph does not necessarily specify the physical **positions** of its nodes
- E.g. these are technically the same graph:



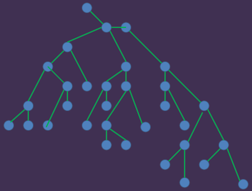
Trees

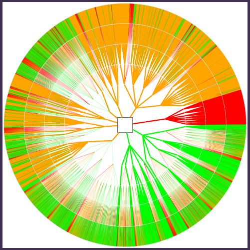
Trees





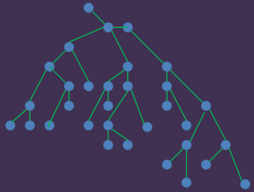
Trees

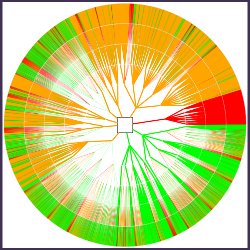




- A **tree** is a special type of directed graph where:

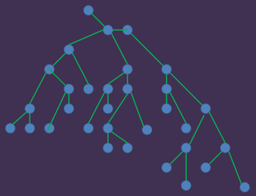
Trees

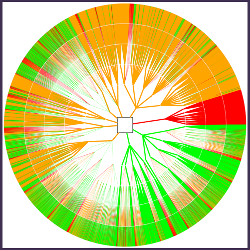




- A **tree** is a special type of directed graph where:
  - One node (the **root**) has no incoming edges

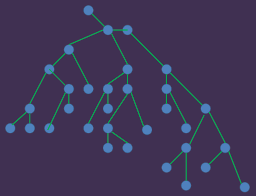
Trees

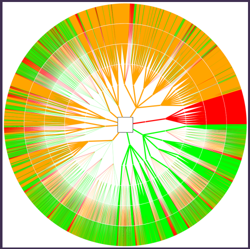




- A **tree** is a special type of directed graph where:
  - One node (the **root**) has no incoming edges
  - All other nodes have exactly 1 incoming edge

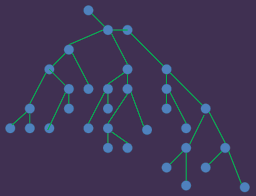
Trees

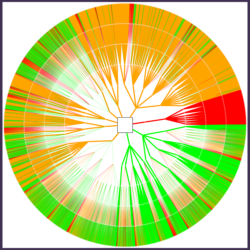




- A **tree** is a special type of directed graph where:
  - One node (the **root**) has no incoming edges
  - All other nodes have exactly 1 incoming edge
- Edges go from **parent** to **child**

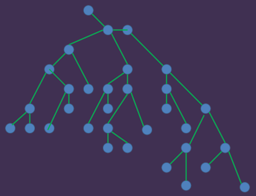
Trees

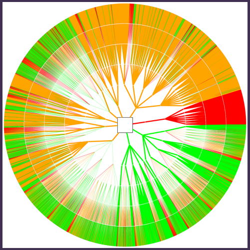




- A **tree** is a special type of directed graph where:
  - One node (the **root**) has no incoming edges
  - All other nodes have exactly 1 incoming edge
- Edges go from **parent** to **child**
  - All nodes except the root have exactly one parent

Trees

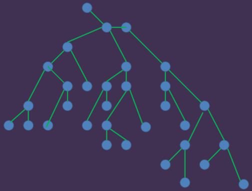


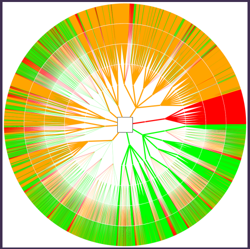


- A **tree** is a special type of directed graph where:
  - One node (the **root**) has no incoming edges
  - All other nodes have exactly 1 incoming edge
- Edges go from **parent** to **child**
  - All nodes except the root have exactly one parent
  - Nodes can have 0, 1 or many children

FALMOUTH UNIVERSITY

Trees





- A **tree** is a special type of directed graph where:
  - One node (the **root**) has no incoming edges
  - All other nodes have exactly 1 incoming edge
- Edges go from **parent** to **child**
  - All nodes except the root have exactly one parent
  - Nodes can have 0, 1 or many children
- Used to model **hierarchies** (e.g. file systems, object inheritance, scene graphs, state-action trees, behaviour trees, ...)

FALMOUTH UNIVERSITY

Tree traversal

Tree traversal

Tree traversal

- **Traversal**: visiting all the nodes of the tree

Tree traversal

- **Traversal**: visiting all the nodes of the tree
- Two main types

Tree traversal

- **Traversal**: visiting all the nodes of the tree
- Two main types
  - Depth first

Tree traversal

- **Traversal**: visiting all the nodes of the tree
- Two main types
  - Depth first
  - Breadth first

Tree traversal

Tree traversal

procedure DEPTHFIRSTSEARCH

FALMOUTH UNIVERSITY

Tree traversal

procedure DEPTHFIRSTSEARCH

let S be a stack

Tree traversal

procedure DEPTHFIRSTSEARCH
    let S be a stack
    push root node onto S

Tree traversal

procedure DepthFirstSearch
    let *S* be a stack
    push root node onto *S*
    while *S* is not empty **do**

Tree traversal

procedure DepthFirstSearch
    let S be a stack
    push root node onto S
    while S is not empty **do**
        pop n from S

Tree traversal

procedure DEPTHFIRSTSEARCH
    let S be a stack
    push root node onto S
    while S is not empty **do**
        pop n from S
        print n

Tree traversal

procedure DEPTHFIRSTSEARCH
    let S be a stack
    push root node onto S
    while S is not empty **do**
        pop n from S
        print n
        push children of n onto S

Tree traversal

procedure DepthFirstSearch
    let *S* be a stack
    push root node onto *S*
    while *S* is not empty **do**
        pop *n* from *S*
        print *n*
        push children of *n* onto *S*
    end while
end procedure

Tree traversal

procedure DEPTHFIRSTSEARCH
    let *S* be a stack
    push root node onto *S*
    while *S* is not empty **do**
        pop *n* from *S*
        print *n*
        push children of *n* onto *S*
    end while
end procedure

procedure BREADTHFIRSTSEARCH

Tree traversal

```
procedure DepthFirstSearch
    let S be a stack
    push root node onto S
    while S is not empty do
        pop n from S
        print n
        push children of n onto S
    end while
end procedure

procedure BreadthFirstSearch
    let Q be a queue
```

Tree traversal

procedure DEPTHFIRSTSEARCH
    let *S* be a stack
    push root node onto *S*
    while *S* is not empty **do**
        pop *n* from *S*
        print *n*
        push children of *n* onto *S*
    end while
end procedure

procedure BREADTHFIRSTSEARCH
    let *Q* be a queue
    enqueue root node into *Q*

Tree traversal

procedure DEPTHFIRSTSEARCH
    let *S* be a stack
    push root node onto *S*
    while *S* is not empty **do**
        pop *n* from *S*
        print *n*
        push children of *n* onto *S*
    end while
end procedure

procedure BREADTHFIRSTSEARCH
    let *Q* be a queue
    enqueue root node into *Q*
    while *Q* is not empty **do**

Tree traversal

procedure DEPTHFIRSTSEARCH
    let *S* be a stack
    push root node onto *S*
    while *S* is not empty **do**
        pop *n* from *S*
        print *n*
        push children of *n* onto *S*
    end while
end procedure

procedure BREADTHFIRSTSEARCH
    let *Q* be a queue
    enqueue root node into *Q*
    while *Q* is not empty **do**
        dequeue *n* from *Q*

Tree traversal

procedure DepthFirstSearch
    let *S* be a stack
    push root node onto *S*
    while *S* is not empty **do**
        pop *n* from *S*
        print *n*
        push children of *n* onto *S*
    end while
end procedure

procedure BreadthFirstSearch
    let *Q* be a queue
    enqueue root node into *Q*
    while *Q* is not empty **do**
        dequeue *n* from *Q*
        print *n*

Tree traversal

procedure DEPTHFIRSTSEARCH
    let *S* be a stack
    push root node onto *S*
    while *S* is not empty **do**
        pop *n* from *S*
        print *n*
        push children of *n* onto *S*
    end while
end procedure

procedure BREADTHFIRSTSEARCH
    let *Q* be a queue
    enqueue root node into *Q*
    while *Q* is not empty **do**
        dequeue *n* from *Q*
        print *n*
        enqueue children of *n* into *Q*

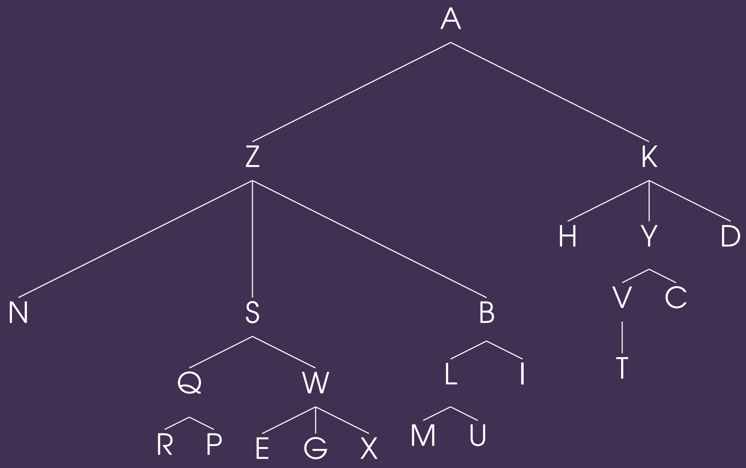
Tree traversal

procedure DEPTHFIRSTSEARCH
    let *S* be a stack
    push root node onto *S*
    while *S* is not empty **do**
        pop *n* from *S*
        print *n*
        push children of *n* onto *S*
    end while
end procedure

procedure BREADTHFIRSTSEARCH
    let *Q* be a queue
    enqueue root node into *Q*
    while *Q* is not empty **do**
        dequeue *n* from *Q*
        print *n*
        enqueue children of *n* into *Q*
    end while
end procedure

Tree traversal example

Socrative `FALCOMPED`



Recursive depth first search

Recursive depth first search

procedure DepthFirstSearch(n)

Recursive depth first search

procedure DEPTHFIRSTSEARCH(n)
    print n

Recursive depth first search

procedure DEPTHFIRSTSEARCH(n)
    print n
    for each child c of n **do**

Recursive depth first search

procedure DEPTHFIRSTSEARCH(*n*)
    print *n*
    for each child *c* of *n* **do**
        DEPTHFIRSTSEARCH(*c*)

Recursive depth first search

procedure DepthFirstSearch(n)
  print n
  for each child c of n **do**
    DepthFirstSearch(c)
  end for
end procedure

Recursive depth first search

```
procedure DepthFirstSearch(n)
    print n
    for each child c of n do
        DepthFirstSearch(c)
    end for
end procedure
```

- Compare to the pseudocode on the previous slide. Where is the stack?

Algorithm strategies

The knapsack problem — informally

The knapsack problem — informally

- You are looting a dungeon in an RPG

The knapsack problem — informally

- You are looting a dungeon in an RPG
- Every item you can pick up has a **weight** and a **value**

The knapsack problem — informally

- You are looting a dungeon in an RPG
- Every item you can pick up has a **weight** and a **value**
- You have a **maximum carry weight**

The knapsack problem — informally

- You are looting a dungeon in an RPG
- Every item you can pick up has a **weight** and a **value**
- You have a **maximum carry weight**
- Which items should you pick up to maximise the total value without exceeding your carry weight?

The knapsack problem — formally

The knapsack problem — formally

- There is a set X of **items**

The knapsack problem — formally

- There is a set X of **items**
- Each item x has a weight $\mathrm{weight}(x)$ and a value $\mathrm{value}(x)$

The knapsack problem — formally

- There is a set X of **items**
- Each item x has a weight $\mathrm{weight}(x)$ and a value $\mathrm{value}(x)$
- There is a maximum weight W

The knapsack problem — formally

- There is a set X of **items**
- Each item x has a weight $\mathrm{weight}(x)$ and a value $\mathrm{value}(x)$
- There is a maximum weight W
- What subset $S \subseteq X$ maximises the total value, whilst not exceeding the maximum weight?

The knapsack problem — formally

- There is a set X of **items**
- Each item x has a weight $\mathrm{weight}(x)$ and a value $\mathrm{value}(x)$
- There is a maximum weight W
- What subset $S \subseteq X$ maximises the total value, whilst not exceeding the maximum weight?
- In other words: find $S \subseteq X$ to maximise

$$\sum_{x \in S} \mathrm{value}(x)$$

subject to

$$\sum_{x \in S} \mathrm{weight}(x) \leq W$$

Algorithm strategies

Algorithm strategies

- Brute force

Algorithm strategies

- Brute force
- Greedy

Algorithm strategies

- Brute force
- Greedy
- Divide-and-conquer

Algorithm strategies

- Brute force
- Greedy
- Divide-and-conquer
- Dynamic programming

Brute force

Brute force

- Try **every possible** solution and decide which is best

Brute force

- Try **every possible** solution and decide which is best

procedure KNAPSACK(X, W)

Brute force

- Try **every possible** solution and decide which is best

procedure KNAPSACK(X, W)

$S_{\text{best}} \leftarrow \{\}$

$v_{\text{best}} \leftarrow 0$

Brute force

- Try **every possible** solution and decide which is best

procedure KNAPSACK(X, W)

$S_{\text{best}} \leftarrow \{\}$

$v_{\text{best}} \leftarrow 0$

for every subset $S \subseteq X$ **do**

Brute force

- Try **every possible** solution and decide which is best

procedure KNAPSACK(X, W)
  $S_{\text{best}} \leftarrow \{\}$
  $v_{\text{best}} \leftarrow 0$
  for every subset $S \subseteq X$ **do**
    if weight$(S) \leq W$ and value$(S) > v_{\text{best}}$ **then**

Brute force

- Try **every possible** solution and decide which is best

procedure KNAPSACK(X, W)
  $S_{\text{best}} \leftarrow \{\}$
  $v_{\text{best}} \leftarrow 0$
  for every subset $S \subseteq X$ **do**
    if weight(S) $\leq W$ and value(S) $> v_{\text{best}}$ **then**
      $S_{\text{best}} \leftarrow S$
      $v_{\text{best}} \leftarrow$ value(S)
    end if

Brute force

- Try **every possible** solution and decide which is best

procedure KNAPSACK(X, W)
  $S_{\text{best}} \leftarrow \{\}$
  $v_{\text{best}} \leftarrow 0$
  for every subset $S \subseteq X$ **do**
    if $\text{weight}(S) \leq W$ and $\text{value}(S) > v_{\text{best}}$ **then**
      $S_{\text{best}} \leftarrow S$
      $v_{\text{best}} \leftarrow \text{value}(S)$
    end if
  end for
  return S_{best}
end procedure

Socrative `FALCOMPED`

Socrative `FALCOMPED`

- If X contains n elements, how many subsets of X are there?
  - Hint: think about constructing a subset as a series of "yes or no" questions

Socrative `FALCOMPED`

- If X contains n elements, how many subsets of X are there?
  - Hint: think about constructing a subset as a series of "yes or no" questions
- Therefore what is the time complexity of the brute force algorithm?

Socrative `FALCOMPED`

- If X contains n elements, how many subsets of X are there?
  - Hint: think about constructing a subset as a series of "yes or no" questions
- Therefore what is the time complexity of the brute force algorithm?
- If we add one element to X, what happens to the running time of the algorithm?

Greedy algorithm

Greedy algorithm

- At each stage of building a solution, take the **best** available option

Greedy algorithm

- At each stage of building a solution, take the **best** available option

procedure KNAPSACK(X, W)

$S \leftarrow \{\}$

Greedy algorithm

- At each stage of building a solution, take the **best** available option

procedure KNAPSACK(X, W)
  $S \leftarrow \{\}$
  for each $x \in X$, in descending order of $\mathsf{value}(x)$ **do**

Greedy algorithm

- At each stage of building a solution, take the **best** available option

procedure KNAPSACK(X, W)
    $S \leftarrow \{\}$
    for each $x \in X$, in descending order of $\mathrm{value}(x)$ **do**
        if $\mathrm{weight}(S) + \mathrm{weight}(x) \leq W$ **then**

Greedy algorithm

- At each stage of building a solution, take the **best** available option

procedure KNAPSACK(X, W)
    $S \leftarrow \{\}$
    for each $x \in X$, in descending order of $\text{value}(x)$ **do**
        if $\text{weight}(S) + \text{weight}(x) \leq W$ **then**
            add x to S
        end if

Greedy algorithm

- At each stage of building a solution, take the **best** available option

procedure KNAPSACK(X, W)
    $S \leftarrow \{\}$
    for each $x \in X$, in descending order of $\mathrm{value}(x)$ **do**
        if $\mathrm{weight}(S) + \mathrm{weight}(x) \leq W$ **then**
            add x to S
        end if
    end for
    return S
end procedure

Greedy algorithm

Greedy algorithm

- Time complexity is dominated by sorting X by value

Greedy algorithm

- Time complexity is dominated by sorting X by value
- The rest of the algorithm runs in linear time

Greedy algorithm

- Time complexity is dominated by sorting X by value
- The rest of the algorithm runs in linear time
- In some problems an appropriately chosen greedy solution is **optimal**

Greedy algorithm

- Time complexity is dominated by sorting X by value
- The rest of the algorithm runs in linear time
- In some problems an appropriately chosen greedy solution is **optimal**
  - A^* pathfinding

Greedy algorithm

- Time complexity is dominated by sorting X by value
- The rest of the algorithm runs in linear time
- In some problems an appropriately chosen greedy solution is **optimal**
  - A^* pathfinding
  - Huffman coding

Greedy algorithm

- Time complexity is dominated by sorting X by value
- The rest of the algorithm runs in linear time
- In some problems an appropriately chosen greedy solution is **optimal**
  - A^* pathfinding
  - Huffman coding
- **However** the greedy solution to the knapsack problem may not be optimal!

Greedy algorithm

- Time complexity is dominated by sorting X by value
- The rest of the algorithm runs in linear time
- In some problems an appropriately chosen greedy solution is **optimal**
  - A^* pathfinding
  - Huffman coding
- **However** the greedy solution to the knapsack problem may not be optimal!
- For example (maximum carry weight is 100)

Greedy algorithm

- Time complexity is dominated by sorting X by value
- The rest of the algorithm runs in linear time
- In some problems an appropriately chosen greedy solution is **optimal**
  - A^* pathfinding
  - Huffman coding
- **However** the greedy solution to the knapsack problem may not be optimal!
- For example (maximum carry weight is 100)
  - Greedy algorithm takes 1 set of horse armour (weight 100, value 500)

Greedy algorithm

- Time complexity is dominated by sorting X by value
- The rest of the algorithm runs in linear time
- In some problems an appropriately chosen greedy solution is **optimal**
  - A^* pathfinding
  - Huffman coding
- **However** the greedy solution to the knapsack problem may not be optimal!
- For example (maximum carry weight is 100)
  - Greedy algorithm takes 1 set of horse armour (weight 100, value 500)
  - ... instead of 100 silver coins (each weight 1, value 10)

Divide and conquer strategies

Divide and conquer strategies

- Break the problem into smaller, easier to solve **subproblems**

Divide and conquer strategies

- Break the problem into smaller, easier to solve **subproblems**
- Requires that the solution to the original problem is composed of the solutions to the smaller problem

Divide and conquer strategies

- Break the problem into smaller, easier to solve **subproblems**
- Requires that the solution to the original problem is composed of the solutions to the smaller problem
- Example from earlier in the module: **binary search**

Divide and conquer strategies

- Break the problem into smaller, easier to solve **subproblems**
- Requires that the solution to the original problem is composed of the solutions to the smaller problem
- Example from earlier in the module: **binary search**
  - Problem: find an element in a list

Divide and conquer strategies

- Break the problem into smaller, easier to solve **subproblems**
- Requires that the solution to the original problem is composed of the solutions to the smaller problem
- Example from earlier in the module: **binary search**
  - Problem: find an element in a list
  - Subproblem: find the element in a list of half the size

Divide and conquer for the knapsack problem

Divide and conquer for the knapsack problem

- Consider an element $x \in X$ with $\mathtt{weight}(x) \leq W$

Divide and conquer for the knapsack problem

- Consider an element $x \in X$ with $\mathtt{weight}(x) \leq W$
- Let X' be X with x removed

Divide and conquer for the knapsack problem

- Consider an element $x \in X$ with $\mathsf{weight}(x) \leq W$
- Let X' be X with x removed
- The solution to the knapsack problem either includes x or it doesn't

Divide and conquer for the knapsack problem

- Consider an element $x \in X$ with $\mathtt{weight}(x) \leq W$
- Let X' be X with x removed
- The solution to the knapsack problem either includes x or it doesn't
- The solution is **either**:

Divide and conquer for the knapsack problem

- Consider an element $x \in X$ with $\text{weight}(x) \leq W$
- Let X' be X with x removed
- The solution to the knapsack problem either includes x or it doesn't
- The solution is **either**:
  - The solution to the knapsack problem on X' with maximum weight W, **or**

Divide and conquer for the knapsack problem

- Consider an element $x \in X$ with $\mathrm{weight}(x) \leq W$
- Let X' be X with x removed
- The solution to the knapsack problem either includes x or it doesn't
- The solution is **either**:
  - The solution to the knapsack problem on X' with maximum weight W, **or**
  - The solution to the knapsack problem on X' with maximum weight $W - \mathrm{weight}(x)$, plus x

Divide and conquer for the knapsack problem

- Consider an element $x \in X$ with $\mathrm{weight}(x) \leq W$
- Let X' be X with x removed
- The solution to the knapsack problem either includes x or it doesn't
- The solution is **either**:
  - The solution to the knapsack problem on X' with maximum weight W, **or**
  - The solution to the knapsack problem on X' with maximum weight $W - \mathrm{weight}(x)$, plus x
- ... whichever has the greater value

Divide and conquer for the knapsack problem

- Consider an element $x \in X$ with $\text{weight}(x) \leq W$
- Let X' be X with x removed
- The solution to the knapsack problem either includes x or it doesn't
- The solution is **either**:
  - The solution to the knapsack problem on X' with maximum weight W, **or**
  - The solution to the knapsack problem on X' with maximum weight $W - \text{weight}(x)$, plus x
- ... whichever has the greater value
- Base case: the solution to the knapsack problem on the empty set **is** the empty set

In other words...

In other words...

- Think about solving the knapsack problem based on the remaining loot and the remaining carry capacity

In other words...

- Think about solving the knapsack problem based on the remaining loot and the remaining carry capacity
- Base case: if you have no carry capacity left, there is nothing to loot

In other words...

- Think about solving the knapsack problem based on the remaining loot and the remaining carry capacity
- Base case: if you have no carry capacity left, there is nothing to loot
- For each piece of loot, try:

In other words...

- Think about solving the knapsack problem based on the remaining loot and the remaining carry capacity
- Base case: if you have no carry capacity left, there is nothing to loot
- For each piece of loot, try:
  - Picking it up and solving the problem with the resulting (reduced) carry capacity

In other words...

- Think about solving the knapsack problem based on the remaining loot and the remaining carry capacity
- Base case: if you have no carry capacity left, there is nothing to loot
- For each piece of loot, try:
  - Picking it up and solving the problem with the resulting (reduced) carry capacity
  - Leaving it and solving the problem with the original carry capacity

In other words...

- Think about solving the knapsack problem based on the remaining loot and the remaining carry capacity
- Base case: if you have no carry capacity left, there is nothing to loot
- For each piece of loot, try:
  - Picking it up and solving the problem with the resulting (reduced) carry capacity
  - Leaving it and solving the problem with the original carry capacity
- Whichever of those two gives the best result, go with it

Divide and conquer for the knapsack problem

procedure KNAPSACK(X, W)

Divide and conquer for the knapsack problem

procedure KNAPSACK(X, W)
    if $X = \{\}$ or $W \leq 0$ **then**
        return $\{\}$
    end if

Divide and conquer for the knapsack problem

procedure KNAPSACK(X, W)
    if $X = \{\}$ or $W \leq 0$ **then**
        return $\{\}$
    end if
    $x \leftarrow$ last element of X

Divide and conquer for the knapsack problem

procedure KNAPSACK(X, W)

  if $X = \{\}$ or $W \leq 0$ **then**

    return $\{\}$

  end if

  $x \leftarrow$ last element of X

  $X' \leftarrow X$ without x

Divide and conquer for the knapsack problem

procedure KNAPSACK(X, W)
    if $X = \{\}$ or $W \leq 0$ **then**
        return $\{\}$
    end if
    $x \leftarrow$ last element of X
    $X' \leftarrow X$ without x
    $S \leftarrow \textsc{Knapsack}(X', W)$

Divide and conquer for the knapsack problem

procedure $\textsc{Knapsack}(X, W)$
    if $X = \{\}$ or $W \leq 0$ **then**
        return $\{\}$
    end if
    $x \leftarrow$ last element of X
    $X' \leftarrow X$ without x
    $S \leftarrow \textsc{Knapsack}(X', W)$
    if $\text{weight}(x) \leq W$ **then**

Divide and conquer for the knapsack problem

procedure KNAPSACK(X, W)
  if $X = \{\}$ or $W \leq 0$ **then**
    return $\{\}$
  end if
  $x \leftarrow$ last element of X
  $X' \leftarrow X$ without x
  $S \leftarrow \textsc{Knapsack}(X', W)$
  if $\text{weight}(x) \leq W$ **then**
    $S' \leftarrow \textsc{Knapsack}(X', W - \text{weight}(x))$
    add x_k to S'

Divide and conquer for the knapsack problem

procedure KNAPSACK(X, W)
- **if** $X = \{\}$ or $W \leq 0$ **then**
  - **return** $\{\}$
- **end if**
- $x \leftarrow$ last element of X
- $X' \leftarrow X$ without x
- $S \leftarrow \text{KNAPSACK}(X', W)$
- **if** $\text{weight}(x) \leq W$ **then**
  - $S' \leftarrow \text{KNAPSACK}(X', W - \text{weight}(x))$
  - add x_k to S'
  - **return** whichever of S, S' has the larger value

Divide and conquer for the knapsack problem

procedure $\textsc{Knapsack}(X, W)$
  if $X = \{\}$ or $W \leq 0$ **then**
    return $\{\}$
  end if
  $x \leftarrow$ last element of X
  $X' \leftarrow X$ without x
  $S \leftarrow \textsc{Knapsack}(X', W)$
  if $\text{weight}(x) \leq W$ **then**
    $S' \leftarrow \textsc{Knapsack}(X', W - \text{weight}(x))$
    add x_k to S'
    return whichever of S, S' has the larger value
  else
    return S
  end if
end procedure

Time complexity

Time complexity

- Each call to KNAPSACK has, in the worst case, **two** recursive calls to KNAPSACK

Time complexity

- Each call to KNAPSACK has, in the worst case, **two** recursive calls to KNAPSACK
- Number of calls is

$$\underbrace{1 + 2 + 4 + 8 + \cdots + 2^i + \dots}_{n \text{ terms}}$$

Time complexity

- Each call to KNAPSACK has, in the worst case, **two** recursive calls to KNAPSACK
- Number of calls is

$$\underbrace{1 + 2 + 4 + 8 + \cdots + 2^i + \ldots}_{n \text{ terms}}$$

- Thus the worst case time complexity is $O(2^n)$ — still exponential!

Time complexity

- Each call to KNAPSACK has, in the worst case, **two** recursive calls to KNAPSACK
- Number of calls is

$$\underbrace{1 + 2 + 4 + 8 + \cdots + 2^i + \dots}_{n \text{ terms}}$$

- Thus the worst case time complexity is $O(2^n)$ — still exponential!
- However in the **average** case many of the calls have only a single recursive call, so this is still more efficient than brute force

Overlapping subproblems

Overlapping subproblems

- Here we end up solving the **same subproblem multiple times**

Overlapping subproblems

- Here we end up solving the **same subproblem multiple times**
- Can save time by **caching** (remembering) these sub-solutions

Overlapping subproblems

- Here we end up solving the **same subproblem multiple times**
- Can save time by **caching** (remembering) these sub-solutions
- This is called **memoization**

Overlapping subproblems

- Here we end up solving the **same subproblem multiple times**
- Can save time by **caching** (remembering) these sub-solutions
- This is called **memoization**
  - **Not** memorization!

Overlapping subproblems

- Here we end up solving the **same subproblem multiple times**
- Can save time by **caching** (remembering) these sub-solutions
- This is called **memoization**
  - **Not** memorization!
- One of several techniques in the category of **dynamic programming**

Dynamic programming for the knapsack problem

procedure KNAPSACK(X, W)
    if KNAPSACK(X, W) has already been computed **then**

Dynamic programming for the knapsack problem

procedure KNAPSACK(X, W)
    if KNAPSACK(X, W) has already been computed **then**
        return previously computed result
    end if

Dynamic programming for the knapsack problem

```
procedure KNAPSACK(X, W)
    if KNAPSACK(X, W) has already been computed then
        return previously computed result
    end if
    if X = {} or W ≤ 0 then
        return {}
    end if
    x ← last element of X
    X′ ← X without x
    S ← KNAPSACK(X′, W)
    if weight(x) ≤ W then
        S′ ← KNAPSACK(X′, W − weight(x))
        add x_k to S′
        cache and return whichever of S, S′ has the larger value
    else
        cache and return S
    end if
end procedure
```

Time complexity

Time complexity

- The running time of a dynamic programming algorithm is limited by the size of the result table — once the table is filled, there is nothing left to do

Time complexity

- The running time of a dynamic programming algorithm is limited by the size of the result table — once the table is filled, there is nothing left to do
- In this case, combinations of X and W

Time complexity

- The running time of a dynamic programming algorithm is limited by the size of the result table — once the table is filled, there is nothing left to do
- In this case, combinations of X and W
- If we always remove the last element of X, then there are $n + 1$ possibilities

Time complexity

- The running time of a dynamic programming algorithm is limited by the size of the result table — once the table is filled, there is nothing left to do
- In this case, combinations of X and W
- If we always remove the last element of X, then there are $n + 1$ possibilities
- Remaining carry weight is an integer between 0 and W — so there are $W + 1$ possibilities

Time complexity

- The running time of a dynamic programming algorithm is limited by the size of the result table — once the table is filled, there is nothing left to do
- In this case, combinations of X and W
- If we always remove the last element of X, then there are $n + 1$ possibilities
- Remaining carry weight is an integer between 0 and W — so there are $W + 1$ possibilities

Socrative `FALCOMPED`

- What is the maximum possible number of entries in the table of intermediate results?

Time complexity

- The running time of a dynamic programming algorithm is limited by the size of the result table — once the table is filled, there is nothing left to do
- In this case, combinations of X and W
- If we always remove the last element of X, then there are $n + 1$ possibilities
- Remaining carry weight is an integer between 0 and W — so there are $W + 1$ possibilities

Socrative `FALCOMPED`

- What is the maximum possible number of entries in the table of intermediate results?
- Therefore what is the time complexity of the dynamic programming algorithm?

Another example of dynamic programming

Another example of dynamic programming

- From the beginning of the lecture:

```
int fibonacci(int n)
{
    if (n <= 2)
        return 1;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}
```

Another example of dynamic programming

- From the beginning of the lecture:

```
int fibonacci(int n)
{
    if (n <= 2)
        return 1;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}
```

- `fibonacci(10)` calls `fibonacci(9)` and `fibonacci(8)`

Another example of dynamic programming

- From the beginning of the lecture:

```
int fibonacci(int n)
{
    if (n <= 2)
        return 1;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}
```

- `fibonacci(10)` calls `fibonacci(9)` and `fibonacci(8)`
- `fibonacci(9)` calls `fibonacci(8)` and `fibonacci(7)`

Another example of dynamic programming

- From the beginning of the lecture:

```
int fibonacci(int n)
{
    if (n <= 2)
        return 1;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}
```

- `fibonacci(10)` calls `fibonacci(9)` and `fibonacci(8)`
- `fibonacci(9)` calls `fibonacci(8)` and `fibonacci(7)`
- `fibonacci(8)` calls `fibonacci(7)` and `fibonacci(6)`

Another example of dynamic programming

- From the beginning of the lecture:

```
int fibonacci(int n)
{
    if (n <= 2)
        return 1;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}
```

- `fibonacci(10)` calls `fibonacci(9)` and `fibonacci(8)`
- `fibonacci(9)` calls `fibonacci(8)` and `fibonacci(7)`
- `fibonacci(8)` calls `fibonacci(7)` and `fibonacci(6)`
- So if we memoize, we can vastly reduce the number of recursive calls

Summary of algorithm strategies

Summary of algorithm strategies

- Brute force

Summary of algorithm strategies

- Brute force
  - Good enough for small/simple problems

Summary of algorithm strategies

- Brute force
  - Good enough for small/simple problems
- Greedy

Summary of algorithm strategies

- Brute force
  - Good enough for small/simple problems
- Greedy
  - Efficient for certain problems, but doesn't always give optimal solutions

Summary of algorithm strategies

- Brute force
  - Good enough for small/simple problems
- Greedy
  - Efficient for certain problems, but doesn't always give optimal solutions
- Divide-and-conquer

Summary of algorithm strategies

- Brute force
  - Good enough for small/simple problems
- Greedy
  - Efficient for certain problems, but doesn't always give optimal solutions
- Divide-and-conquer
  - Good if the problem can be broken down into simpler subproblems

Summary of algorithm strategies

- Brute force
  - Good enough for small/simple problems
- Greedy
  - Efficient for certain problems, but doesn't always give optimal solutions
- Divide-and-conquer
  - Good if the problem can be broken down into simpler subproblems
- Dynamic programming

Summary of algorithm strategies

- Brute force
  - Good enough for small/simple problems
- Greedy
  - Efficient for certain problems, but doesn't always give optimal solutions
- Divide-and-conquer
  - Good if the problem can be broken down into simpler subproblems
- Dynamic programming
  - Makes divide-and-conquer more efficient if subproblems often reoccur

Workshop