COMP110: Principles of Computing

10: Algorithm Strategies

Worksheets

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

FALMOUTH UNIVERSITY

Recursion

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);
}
```

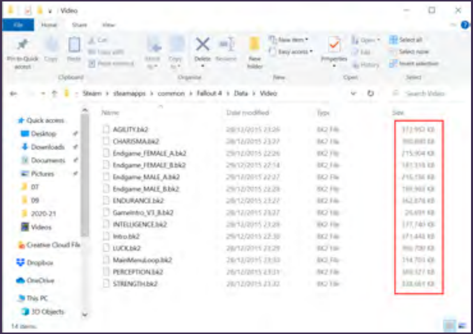
- Recursive functions need a **base case** where they stop recursing, otherwise they will go **forever**

Thinking recursively

- 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

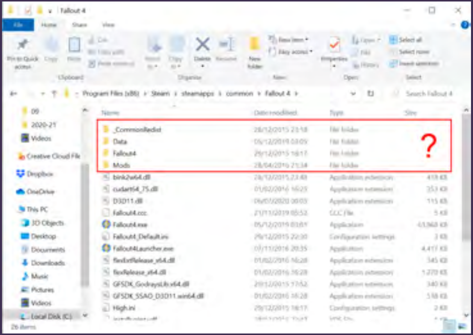
- Suppose we want to find the total size of all files in a folder and its subfolders



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

Example: file sizes

- What if the folder contains subfolders?



- 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

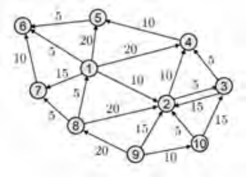
procedure CALCULATEFOLDERSIZE(*folder*)
    totalSize $\leftarrow 0$
    for each *item* in *folder* **do**
        if *item* is a file **then**
            totalSize $\leftarrow$ *totalSize* $+$ GETFILESIZE(*item*)
        else if *item* is a folder **then**
            totalSize $\leftarrow$ *totalSize* $+$ CALCULATEFOLDERSIZE(*item*)
        end if
    end for
    return totalSize
end procedure

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

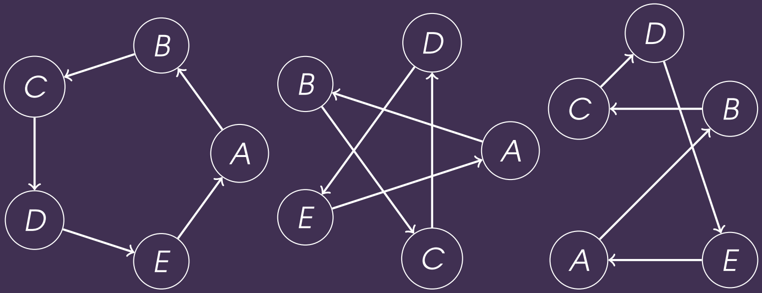
Graphs



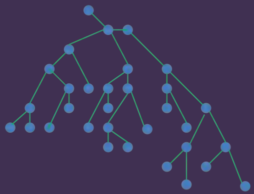
- A **graph** is defined by:
  - A collection of **nodes** or **vertices** (points)
  - A collection of **edges** or **arcs** (lines or arrows between points)
- Often used to model **networks** (e.g. social networks, transport networks, game levels, automata, ...)
- **Directed** graph: edges are arrows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

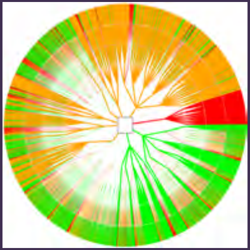
Drawing graphs

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



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, ...)

Tree traversal

Tree traversal

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

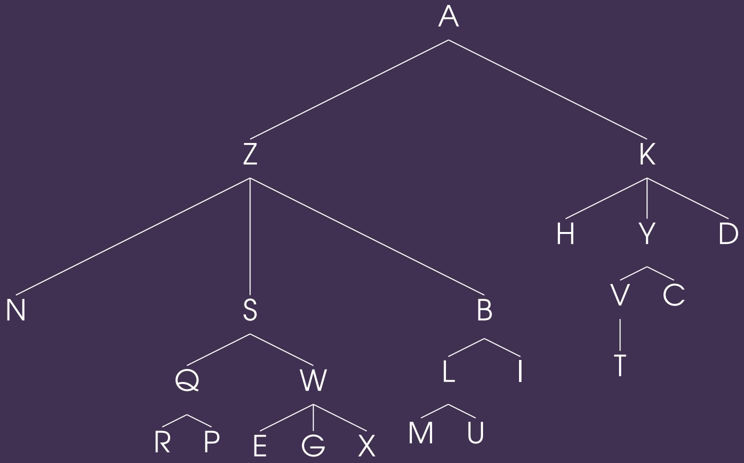
Tree traversal

```
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

```
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

- 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

- 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

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

Brute force

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

```
procedure KNAPSACK(X, W)
    S_best ← {}
    v_best ← 0
    for every subset S ⊆ X do
        if weight(S) ≤ W and value(S) > v_best then
            S_best ← S
            v_best ← value(S)
        end if
    end for
    return S_best
end procedure
```

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

- 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

- 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

- 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

- 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
- Base case: the solution to the knapsack problem on the empty set **is** the empty set

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

- 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

- 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
        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

- 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

- 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

- 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

FALMOUTH UNIVERSITY

Workshop