Examinable material for COM2001: Advanced Programming Techniques – James Marshall

This document is intended to cover all the examinable material for the COM2001 second semester exam, as specified in the following table:

Original here: http://staffwww.dcs.shef.ac.uk/people/J.Marshall/apt/assignments/exam_guide.pdf

1	• How to write evaluation traces for recursive functions
2	• How to work with asymptotic complexity notations • Definition of worst-case vs average-case complexity
3	• How to design divide-and-conquer algorithms • How to use recurrence trees
4	• How to solve arithmetic series • How to solve geometric series • How to analyse algorithms with the Master Theorem
5	• Definition of defined and undefined values • How to use proof by induction (standard) • How to use proof by induction (structural)
6	• How to use proof by induction (strong) • How to use proof by induction in imperative languages
7	• How to work with axiomatic specifications of abstract data types • How to implement abstract data types in Haskell
8	• How to show completeness of abstract data type specifications
9	• Definition of problems efficiently solvable by dynamic programming

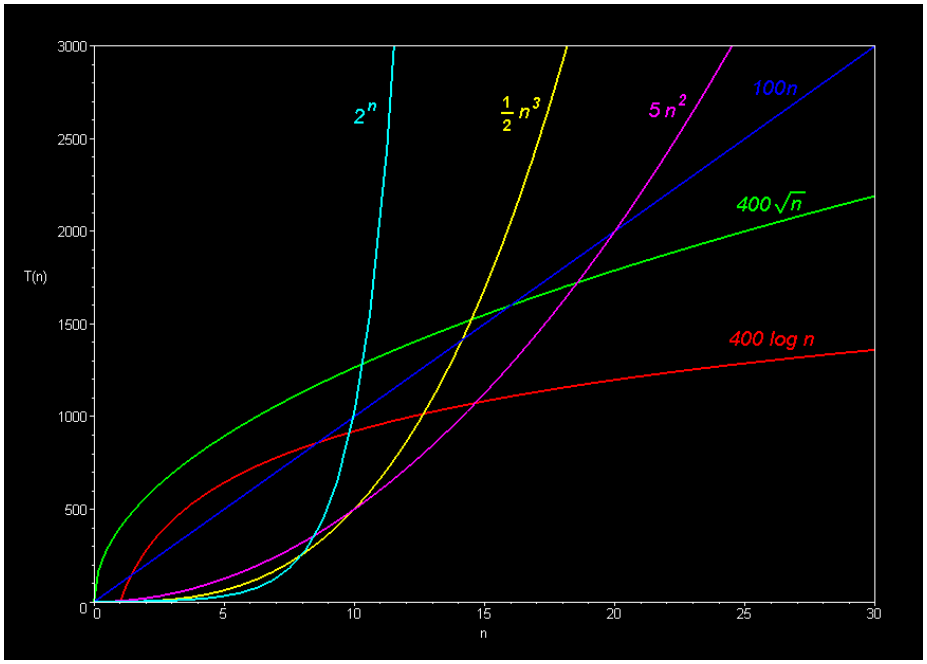
1 EVALUATION TRACES FOR RECURSIVE FUNCTIONS (SEE SECTION 3.2)

The evaluation trace of a recursive function is the sequence of steps taken to evaluate a recursive function. At each step, the problem is split into smaller problems of the same type. Evaluating an expression means expanding it into the combination of smaller problems. For example evaluating **Fib(4) -> Fib(3) + Fib(2)** etc. as in *section 3.2.*

2 COMPLEXITY

The Time-Complexity of an algorithm tells us the **running time** of an algorithm **as a function of the input size** to the algorithm.

Here are graphs of some possible time complexity functions [1]:



'Asymptotic' means the *actual* running time isn't necessarily known, but we do know an 'asymptote' of the running time. Which means we have an estimation for the running time which gets more accurate/relevant with larger input sizes.

An upper/lower bound gives us a maximum/minimum running time for an algorithm, again only for sufficiently large inputs. For example an algorithm with input size **N** could - for sufficiently large **N** - always run *faster (i.e. in less time)* than $\mathbf{2^N}$**.** This would be an upper bound of $\mathbf{2^N}$ to the time complexity of the algorithm.

2.1 SYNTAX FOR UPPER AND LOWER BOUNDS

When specifying the time complexity of an algorithm, we use **Big-O notation** (or big-omega, big-theta).

O(log(n)) means **log(n)** is the *asymptotic upper bound* for the *time complexity* of an algorithm with **input-size n**.

$$O(n) = upper\ bound$$

$$\Omega(n) = lower\ bound$$

$$\Theta(n) = upper\ \&\ lower\ bound$$

The function on the inside of the big-O notation should only contain the *highest order term* within the time complexity. For example, rather than writing **O(2n + 3)** you should write **O(n)**, and rather than writing **O($2n^2$ + 100n + 9)** you should write **O(n^2).** Note that we have to remove both lower order terms and any constant multiples of the highest order terms, for example we got rid of **+ 9**, **+ 100n** and the **2 *** multiple of **n^2**.

2.2 Complexity Classes

Strictly **O(f(n))** refers to a *complexity class.* This is the set of functions that have **f(n)** as a valid upper bound. We therefore use set notation to talk about complexities. A good illustration of this is from the mock assignment:

$$O(1) \subset O(log(n)) \subset O(n) \subset O(n\,log(n)) \subset O(n^2) \subset O(2^n)$$

The complexity classes to the left are *subsets* of the complexity classes to the right, because a larger upper bound is still a valid upper bound to a function that could have a lower one. i.e. **f(n) = 2^n** is a valid upper bound for **f(n) = n**, which is also a valid upper bound for **f(n) = 1.** The opposite of this is true if we were talking about *lower bounds***.**

2.3 Worst Case vs Average Case

The worst case is simple: it's the time complexity for the input which takes the longest time. For example for a sorting algorithm taking an input of 1000 integers, it might take no time at all if the list is already sorted. The worst case running time for an input of 1000 integers would be if the integers were ordered such that no other ordering would take a longer time to sort with that algorithm.

The average case is more difficult, as it requires knowledge about the expected input. To work out the average case, work out the time complexity for the different inputs, and average these times while weighting them for their expected probability of occurrence.

3 DIVIDE AND CONQUER, RECURRENCE

3.1 DIVIDE AND CONQUER

A divide and conquer algorithm is one that solves a problem in the following way:

1. Split the problem into a few *smaller & simpler* sub-problems.
2. Solve each of the sub-problems in turn.
3. Combine the results to find the solution to the original problem.

It is crucial that the sub-problems are in some way easier to solve, otherwise the technique is pointless. For example we can write an algorithm to calculate factorials like follows:

1! = 1

and

N! = (N + 1)! / (N + 1) *written in terms of the factorial above*

Or

N! = (N – 1)! * N *written in terms of the factorial below*

They're both correct, but computing the factorial above is harder than computing the factorial below, so the second algorithm is the only one that would be useful.

3.2 RECURRENCE AND RECURRENCE TREES

If the sub-problems are of *exactly the same type* as the original problem, then the result will be a recursive algorithm. I.e. an algorithm which is defined in terms of itself.

The above factorial definition only uses one instance of itself in its definition. Iteratively expanding the definition for **5!** Would look as follows:

5!

*4! * 5*

*3! * 4 * 5*

*2! * 3 * 4 * 5*

*1! * 2 * 3 * 4 * 5*

*1 * 2 * 3 * 4 * 5*

The second classic example of recursion is the Fibonacci sequence:

Fib(N) = Fib(N – 1) + Fib(N – 2)

Fib(1) and Fib(2) = 1

Iteratively expanding **Fib(5)** would look as follows:

Fib (5)

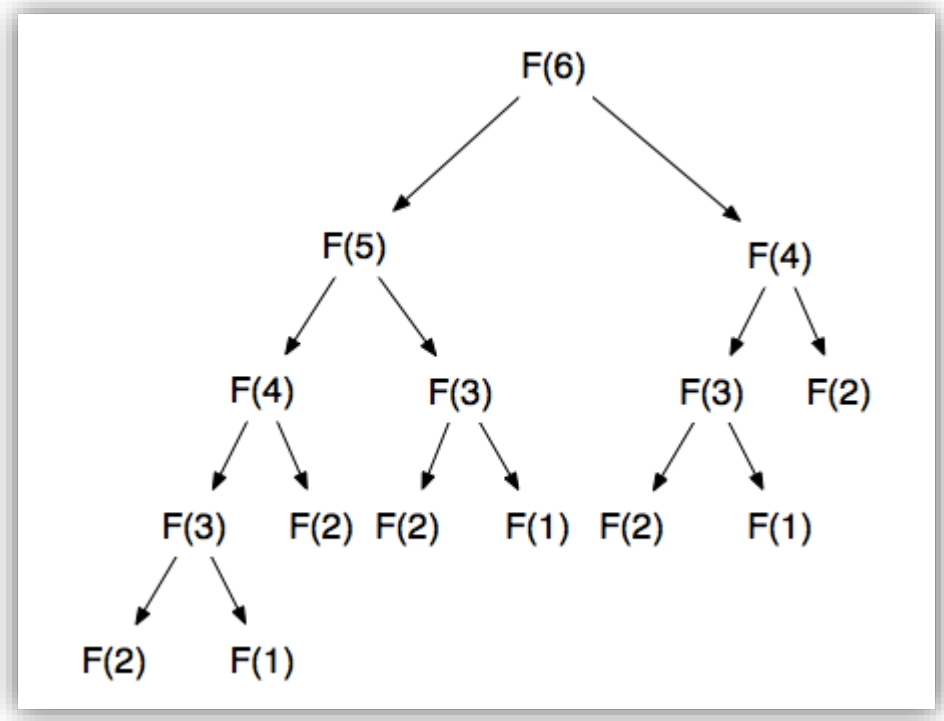
Fib(4) + Fib(3)

(Fib(3) + Fib(2)) + (Fib(2) + Fib(1))

((Fib(2) + Fib(1)) + 1) + (1 + 1)

((1 + 1) + 1) + (1 + 1)

This is called the recurrence tree. The number of times the tree branches at each node is the number of smaller sub-problems the divide and conquer approach has used, and is the number of recurrences at each stage. Recurrence trees may best be drawn as actual trees with lines connecting successive expansions, as in the following image [2]:



4 SUMMING FINITE SERIES AND THE MASTER THEOREM

This section explains how to calculate the asymptotic time complexity of recursive algorithms using the "Master Theorem" or summing of finite series.

4.1 SUMMING ARITHMETIC SERIES

An Arithmetic sequence is a sequence of numbers where each number is a constant number larger than the previous. For example **2, 5, 8, 11, 14** … each number is **+3** larger than the last.

The *closed-form* solution for summing a finite number of terms in an arithmetic series is:

$$\frac{n(a_1 + a_n)}{2}$$

Where **n** is the number of terms being added, $\mathbf{a_1}$ is the first term and $\mathbf{a_n}$ is the last term. For the sequence **2, 5, 8, 11, 14** the answer would be **5 * (2 + 14) / 2 = 40**

4.2 SUMMING GEOMETRIC SERIES

A Geometric sequence is a sequence where each term is a constant multiple of the previous, for example **1, 2, 4, 8, 16, 32** … each number is **twice as large** as the last.

The closed-form solution for summing a finite number of terms in a geometric series is:

$$a_1 \frac{x^n - 1}{x - 1}$$

Where **n** is the number of terms, **x** is the ratio between successive terms, and $\mathbf{a_1}$ is the first term. For the sequence **4, 12, 36, 108** the answer would be **4 * (3^4 – 1) / (3 – 1) = 160.**

4.3 USING SERIES FOR FINDING TIME-COMPLEXITY

When the running time for an algorithm can be specified as a summation of terms (see 2d and 2e in the mock assignment) the formulas above can be used to find the closed form solution for the time complexity, which can then be simplified into **Big-O** notation.

4.4 The Master Theorem

The master theorem is a 'cookbook' method for finding the time complexity of an algorithm. This means it *should* be usable without any understanding of how it works.

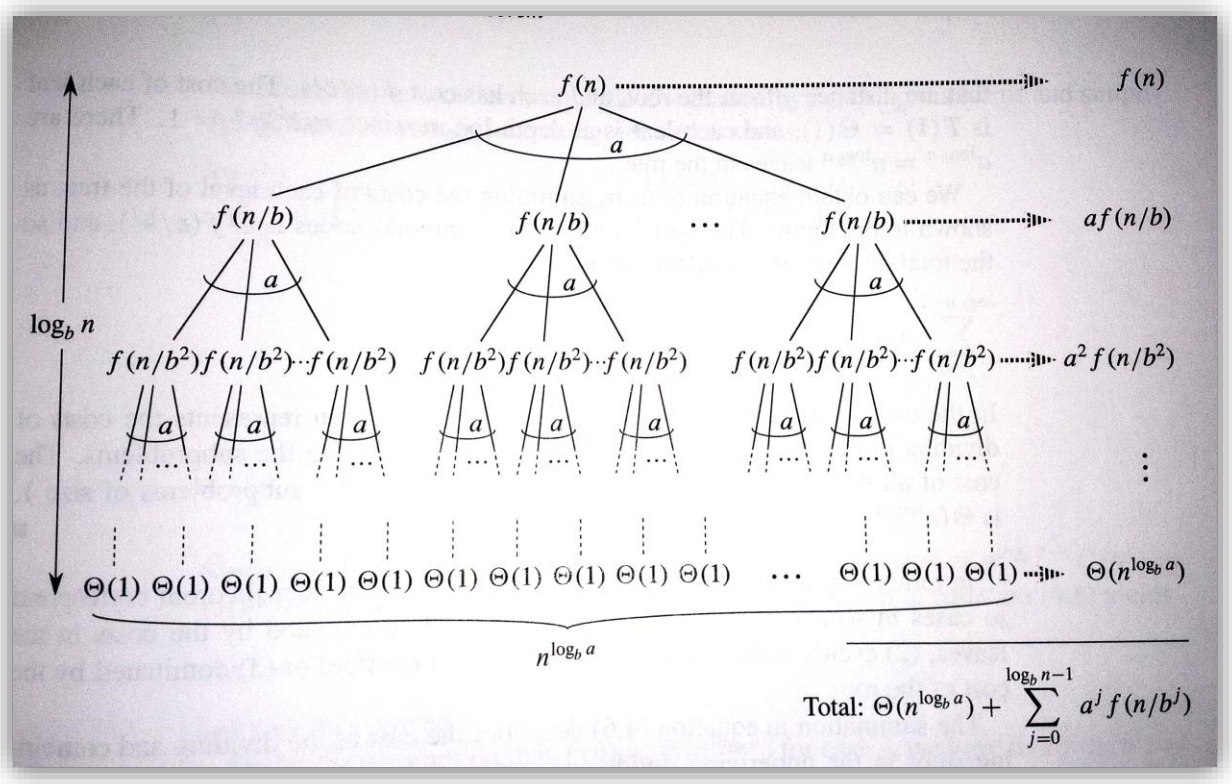
If a function has time complexity **T(n)** …

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Meaning at each stage, the problem (size **n**) is split into **a** parts, each of size **n/b** which are each solved with a time complexity of **T(n/b)** and combined with a cost of **f(n)**. So **a*T(n/b)** is the time taken to solve all the sub-problems, and **f(n)** is the extra work needed to split the problem down and put the solutions back together.

The diagram below [3] shows a general breakdown of the time complexity of an algorithm given the equation above. There are a couple of important things to note:

1. Each terminating node (leaf) in the recursion tree is completed in constant **O(1)** time, and there are **n^log$_b$(a)** of them, meaning the leaves together contribute **O(n^log$_b$(a))** time.
2. The summation of all the **f(n)**'s from the rest of the nodes (down the right hand side of the diagram) add up to make the rest of the formula, as you can see in the bottom right of the diagram.



The Master theorem looks at the values for **a**, **b** and **f(n)** and derives which part of this general equation is the highest order part – i.e. which part is most significant as **n** becomes large. There are three possibilities:

1. Most of the work is done at the leaves when **n** is large. This means the algorithm has complexity:

$$\Theta(n^{log_b a})$$

   This is always the case if:

$$f(n) = O(n^{log_b a - \varepsilon})$$

   Where ε is some positive constant. This means the time complexity of **f(n)** is polynomially smaller than the time complexity of all the work done at the leaves.

2. The work done at the leaves and at each node have similar time-complexities (i.e. neither the leaves nor nodes completely dominate the running time for sufficiently large **n**). In this case the time complexity is:

$$\Theta(n^{\log_b a} \log n)$$

   This is always the case if:

$$f(n) = \Theta(n^{log_b a})$$

   This means that **f(n)** has the same time complexity as the work done at the leaves.

3. The work done at the **root** dominates as **n** becomes large. Time complexity is:

$$\Theta(f(n))$$

   This is always the case if both:

$$f(n) = \Omega(n^{log_b a + \varepsilon})$$

and

$$a * f\left(\frac{n}{b}\right) \leq c * f(n)$$

   For some constant **c < 1**, some positive constant ε and for sufficiently large **n**. This means two things: That the work done at the nodes will dominate over the work done at the leaves, and that the work done at the root node will dominate over the work done at all the other nodes.

To apply the master theorem, just find values for **a**, **b** and **f(n)** and check which of the conditions hold from 1, 2 or 3. Once you have found this, the time complexity of the algorithm is known. If none of the conditions hold, or the algorithm's running time can't be written in terms of **a**, **b** and **f(n)**, the master theorem doesn't apply. See answers to parts **3a – 3i** in the mock assessment for examples.

5 PROOF BY INDUCTION (STANDARD AND STRUCTURAL)

5.1 DEFINED AND UNDEFINED RESULTS FOR RECURSIVE FUNCTIONS

For a recursive Haskell function, the result of calling a function is defined if the recursion will terminate at some point and return a result. The value is undefined if the recursion continues for ever, for example see the recursive definitions of factorial given in **3.2** of this document.

5.2 WHAT IS STANDARD/STRUCTURAL INDUCTION?

Standard induction proves that some property holds over all the natural numbers. For example if we can prove that **P(0)** holds, and **P(n) => P(n + 1)** for all **n**, we have proved that **P(n)** holds for all natural numbers **n**.

Structural induction does the same, but for lists of length **n**. If we can prove that **P([])** holds, and **P(t) => P(h:t)** for all lists **t** and elements **h**, we have proved that **P(lst)** holds for all lists **lst**.

Proofs using induction (standard and structural) should be formatted as follows:

1. **Proof outline:** State your proof goal, base case and induction stage clearly in Haskell syntax, labelling them as such.
2. **Prove base case:** Usually the base case will be the same as the final proof goal, but with some variable substituted with **0** (for standard induction) or **[]** (for structural induction).
3. **Prove the induction case:** In this part, you can assume the property holds for **n** and you must show it holds for **n + 1** (or some list **x** and then **h:x** for structural induction).
4. **State clearly that these three proofs together prove that P(x) holds for all x. QED.**

At each stage (1, 2 and 3) the proof is completed by showing that the left and right hand side of the equations are equal by iteratively substituting and manipulating them using the supplied Haskell code. At each substitution, you should label the line with the substitution you've used (for example **++.1** or **reverse.2**).

There is a very thorough and helpful guide on all of this, with lots of examples, in **Chapter 8 of Haskell – The craft of functional programming**. Available in the IC and St. George's.

6 STRONG INDUCTION AND IMPERATIVE PROOF

6.1 STRONG INDUCTION

In standard induction, the induction step proves that **P(n) implies P(n + 1)** for all **n**. Sometimes assuming **P(n)** is not enough to prove **P(n + 1)**.

Strong induction assumes **P(n)** is true along with **P(n – 1), P(n – 2)** and **P of all values between 0 and n**. So with strong induction we show that **(P(K) for all 0 <= K <= n) implies P(n + 1).**

Strong induction, despite its name, is just as powerful as weak induction. There is nothing that can be proven with the strong version that can't be proven with the weak version. However using strong induction allows for more concise proofs for things like the Fibonacci sequence where the definition of **Fib(n)** includes **Fib(n – 2)** and not just **Fib(n – 1)**.

6.2 IMPERATIVE PROOFS – LOOP INVARIANTS

Imperative programs are programs written as a sequence of steps. Most common languages are imperative at their core, i.e. javascript/java/ruby etc. although they may have 'functional' type features.

In Marshall's notes, he gives one example of proving an imperative program correct, and it is proven informally on a pseudo-code implementation of insertion sort. This example is paraphrased from chapter 2 of CLRS.

Proofs that an imperative program works need to show that some *invariant* holds throughout the running time of the algorithm/loop. The invariant, by the time the loop or program terminates, should be some desirable characteristic, for example that a list is sorted.

Structure your imperative proofs as follows:

1. **State the loop invariant:** This is some property of the *state* of the program that you want to remain true throughout its running. This might be given to you in the question.
2. **Initialisation:** Show that the property is true before the program or loop has started running. This should be trivial; just look at what the invariant applies to at the beginning of the program (maybe an empty list, or the number 0 etc.) and show that it is true.
3. **Maintenance:** Prove that the contents of the loop will not make the invariant false if it is true at the start of the loop. I.e. true at start => true at end.
4. **Termination:** State how the loop invariant, which is still true due to the above two conditions, shows your program's correctness at termination. The example in the notes states that the loop invariant is a sorted sub-list of the input list, which at the point of program termination is the whole input list, now completely sorted.

For a couple more examples of informal proofs of imperative programs, see **chapter 2 of CLRS**.

Mike Stannet's notes introduce **Floyd–Hoare logic** which is a more formal logical system for proving correctness of imperative programs, although this isn't mentioned in Marshal's notes. Any question about imperative proof will probably require an informal proof.

7 Axiomatic Specifications and Abstract Data Types

7.1 Abstract Data Types

Abstract Data Types (ADTs) are a concept in computer science independent of any language. There are no 'ADT' keywords or constructs within Haskell. Instead it is said that an ADT (usually some mathematical model of a data structure) can be implemented in Haskell using **modules**, which *are* a specific Haskell structure. It is said that Haskell modules allow ADT's to be built in Haskell.

The characteristic feature of an ADT is that it can (and should) be used without knowledge of its implementation – this is also called information hiding/abstraction, as the information about the ADT's implementation is hidden/abstracted away from the ADT's user. Haskell modules allow this.

To start a Haskell module

```
module ModuleName (MemberOne, MemberTwo, MemberThree)
      where -- followed by definitions for each member
```

When 'ModuleName' is imported, we won't be able to depend on the specific implementation within the module.

For a couple of examples, see Marshall's Set and Stack modules:

Stack: http://staffwww.dcs.shef.ac.uk/people/J.Marshall/apt/haskell/Stack.hs

Set: http://staffwww.dcs.shef.ac.uk/people/J.Marshall/apt/haskell/Set.hs

7.2 Axiomatic Specifications of ADTs

An ADT can be specified axiomatically. This specification will be a collection of relationships between functions which need to hold true for any implementation. For example, part of the specification of Marshall's Stack ADT is:

```
isEmpty emptyStack == True -- (specIsEmpty.1)
```

Any Haskell implementation of this Stack ADT must satisfy this condition, and all other axioms.

Remember that these axioms, although they're written in Haskell syntax, exist outside any Haskell implementation. They're abstract mathematical descriptions of a data structure that could have many implementations. You may be asked to prove a specific implementation satisfies a certain collection of axioms. Proving this is exactly the same as any other proof of program correctness, so see **section 5** of this document.

8 COMPLETENESS AND PROOFS WITH ADT'S

Completeness, in this context, is a property of the axiomatic specification of an ADT and **not** the implementation. It is also not a *formal* property, as an axiomatic specification is only complete relative to an intuitive (i.e. informal) idea of how the ADT should behave. To say that an axiomatic specification is *complete* is to say that it aligns *completely* with your intuitive understanding, without the need for any additional axioms.

This means an axiomatic specification can be said to be *incomplete* if there is a property which cannot be proven/derived from the axioms, but which still should 'intuitively' hold.

You will need to use the same methods of proof described in section 5 to prove that an axiomatic spec is satisfied by a particular implementation.

9 DYNAMIC PROGRAMMING

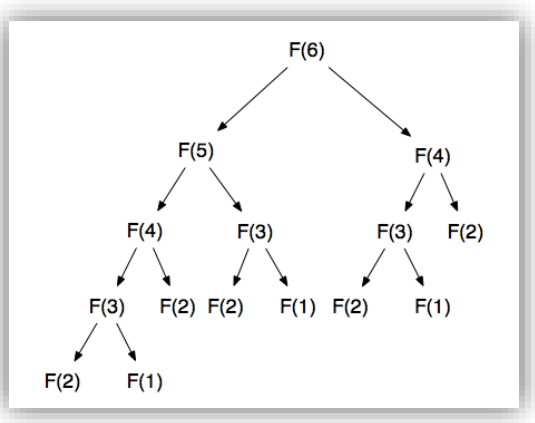
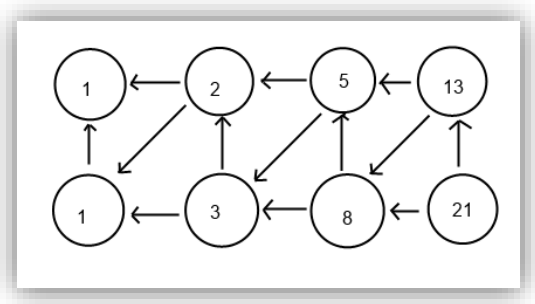
A problem is efficiently solvable with dynamic programming if it satisfies the following 2 conditions:

1. **Optimal Substructure:** Finding the optimal solution to the problem involves finding the optimal solution to some other sub-problems.
2. **Overlapping sub-problems:** A solution to a sub-problem can be used by multiple other problems, and the sub-problems that need to be solved for a specific problem will share other sub-problems. An example of this is the Fibonacci sequence: Fib(10) = Fib(9) + Fib(8), and the two sub-problems of Fib(9) and Fib(8) overlap because they both require working out Fib(7) etc.

Together, these two conditions allow memoization to be effective. Memoization effectively adds a cache to the function, so it doesn't solve from scratch a particular sub-problem twice.

9.1 SUB-PROBLEM GRAPHS

Sub-problem graphs are like recurrence trees but with memoization, so no sub-problem is worked out twice from scratch. Compare the following two diagrams for calculating the Fibonacci numbers:

Simple recurrence (repeated calculations) [3]	With dynamic programming (memoization) [4]
	

On the left hand side, F(4) is calculated twice, once on the left and once on the right. On the right hand side (with dynamic programming) F(4) – or 3 in the diagram – is only calculated once for all larger Fibonacci numbers. The diagram on the left is a recurrence tree, and the diagram on the right is a sub-problem graph.

10 REFERENCES/SOURCES

[1] Taken from http://science.slc.edu/~jmarshall/courses/2002/spring/cs50/BigO/ without permission… Yes that course is also taught by a J. Marshall. Shock horror!

[2] Stolen shamelessly from http://blog.marquiswang.com/2010/02/20/fibonacci-functions-and-dynamic-programming/ without permission.

[3] Photo taken by me from Introduction to Algorithms (CLRS), page 77 in 2001 version.

[4] Drawn by me in Paint.NET