Diploma in Computer Science
Computer Science Tripos II(G)

Introduction to Algorithms

http://www.cl.cam.ac.uk/Teaching/2004/IntroAlgs

Martin Richards mr@cl.cam.ac.uk

Michaelmas 2004

These notes are based on those of Dr A.C. Norman and Dr A. Mycroft

(Version October 7, 2004)

1 Introduction

This is a set of four lectures aimed for the Diploma in Computer Science and the Computer Science Tripos (Part 2 General) classes. It leads in to the "Data Structures and Algorithms" course that is timetabled immediately after it. The DS&A course is also attended by CST 1B students, who had a couple of courses on programming last year as well as one on Discrete Mathematics. The programming side of things is covered to a large extent by the fact that the Diploma class gets taught Java early in the term, thus this course provides survival training with regard to the mathematics that the DS&A course needs to rely on. Those Diploma students who have just completed a first degree in mathematics (elsewhere or here, and possibly including Part 3) might reasonably collect a copy of these notes, observe the topics covered and not worry about attending these four lectures[1].

Later in the year there is a course that has "Discrete Mathematics" as its title. There may be a small amount of overlap between this course and that one, but I do not mind! Firstly because some things can usefully be said twice and make more sense the second time, and secondly because the coverage I give here will be somewhat compressed and will relate only the the DS&A course, while Discrete Mathematics has other applications in Computer Science.

This set of four lectures is not directly examined. Some of the concepts introduced may, however, help in answers to DS&A questions, and it may give a preview of Discrete Maths and hence make that easier to cope with.

2 Teaching and Learning Guide

[This is a Computer Laboratory mandated section of all lecture courses.]

Since this course is not directly examined (its purpose is mainly to (re-) introduce discrete mathematics needed for the 'Data Structures and Algorithms Course') there is no obvious source of questions. For additional problems I would suggest that you turn to the Teaching and Learning Guides issued by the (two different) Discrete Mathematics courses given as part of the Diploma/Part 2 General and as part of the Computer Science Tripos (Part 1A).

The general areas covered in the four lectures are

- Sets, functions.
- Relations, graphs; induction.
- Combinations and permutations.
- $O(f)$ notation, sorting as example algorithm.

[1]However note that before dropping any **other** lectures you ought to discuss your plans with your Director of Studies.

If you want more reading on this sort of material, I recommend “Concrete Mathematics” by Knuth, Graham and Patashnik. It obviously contains **much** more that can fit in my four lectures, but a great deal of what it contains could find direct use somewhere in a computer science course. The Schaum-series book ‘Discrete Mathematics’ contains some 700 worked problems and is in general quite useful.

3 Proofs and Induction

A significant issue in designing proper computer procedures to solve problems is that of proving that they **always** work. There is a very large body of empirical evidence that just writing a program and testing it (even testing it on a very large number of examples) is not sufficient: formal proof is necessary. Examples given in this course will not all have a very obviously computerish feeling—I hope that when the DS&A course follows it you will see some of the applications of notation introduced here.

The first piece if notation to introduce here is the symbol “$\Rightarrow$” which I will pronounce “implies”. The idea is that if you know that the thing on the left is true then you can deduce that the one on the right is. A major use of this is in the following construction:

If you know both A and $A \Rightarrow B$
Then you may deduce B

whatever the statements A and B are. Eg:

All cats are animals
and *Arthur is a cat*
therefore
Arthur is an animal

To bring the example better into line with the original formalism it is perhaps helpful to see “All cats are animals” as shorthand for standing for all possible statements of the form “x is a cat $\Rightarrow$ x is an animal”, with arbitrary words (including in particular including “Arthur”) substituted for the marker x.

What you find is that the more you strain to make real-world examples precise the more murky they appear. Some of this is a direct reflection of the flexibility and imprecision of natural language. (A pretty example of the imprecision of English here is the statement ‘everyone loves someone’. Does this mean that every person has there own little favourite(s)? Or that there is some shared someone which everyone admires to the point of loving?) Often things that start off mathematical in style fit into the structure of formal proofs more easily.

The notation “$A \Rightarrow B$” can be read as “If A then B” or “B is true if A is” or “whenever A is true it happens that B is true”. So far we have use this in cases where we have some way of knowing that A is true. Another use of the same rule is when we can tell (from some other source) that B is false. It is then valid to deduce that A must be false also. For instance given that all cats are animals, knowing that the statement “the moon is an animal" is false assures us that the moon is not a cat. Wow! If you were a class of mediaeval philosophers you would know these rules as *modus ponens* and *modus tolens*, and we could talk about deduction using them as syllogisms. I hope the rules look like natural common sense, but it is worth noting that in common speech (especially of a political nature!) the rules are very often not adhered to. As an exercise try to spot examples where people try to carry forward an argument but break the formal rules of logic. Eg modified from a Thurber story

People who eat carrots and have long ears cause earthquakes
and *there was just an earthquake*
therefore
It was caused by the rabbits
therefore *We wolves are justified in taking them into protective custody*
...to protect ourselves

Some people would consider that the above line of argument illustrates various other bad forms of deduction, such as starting the argument off with an assertion that is false, and slightly adjusting the interpretation of "protect" part way through (etc).

On occasions it will also be useful to write things the other way around as in "$B \Leftarrow A$", which can perhaps unexpectedly be read as "A only if B". To see how, observe that A being true while B is false contradicts the implication. Finally $A \Leftrightarrow B$ is a shorthand for having both $A \Rightarrow B$ and $A \Leftarrow B$, and to prove it you normally need to prove both the **if** and the **only if** parts—often separate proofs will be required for these two parts. A very common confusion is to mix up whether you are in the process of proving the $\Rightarrow$ or $\Leftarrow$ part.

Often the sorts of things you want to prove in computer science (and mathematics) will be general statements true for **all** cases of some condition. One approach to this is known as *proof by induction*. It comes in a number of variants, so I will deal with the simplest one first:

Suppose I want to prove that some result is true for all integer values of n, ie $n = 1$, $n = 2$, $n = 3$, ... then I can start by proving the *base case* which will be the first one, $n = 1$ (or $n = 0$, since computer scientists often start counting at zero), and then showing that if the result is true for some particular value of n then it must also be true for the next step up $(n + 1)$.

As an example, consider the Tower of Hanoi problem—you have a set of graduated discs and three pegs. A larger disc may never be piled on top of a smaller one, and the discs live on the pegs. In a single move you may move one disc from the top of the pile on one peg to the top of the pile on another peg, but you must keep to the rule that large discs may never be placed above small ones. The result to be proved is that however many discs there are it is possible (starting with all discs piled in order on one peg) to make a sequence of moves that end up with the discs on another nominated peg.

The proof by induction goes as follows

Base case: With one disc it is easy - just move it!

Induction step: Suppose we can achieve the desired effect with k discs, and now we have $k+1$. Calls the discs X, Y and Z and suppose all discs start on X and are to be moved to Y. Observe that if we just ignore the bottom (largest) disc the remaining k discs are subject to the same rules of the Hanoi game, so by our induction hypothesis there is a sequence of moves that ends up with all of them on peg Z. From this state it is possible to move the largest disc from X to Y. For the remainder of the sequence again the largest disc can be ignored, and again moving the k discs from Z to Y can be done. In the end all discs are on peg Y in the desired order, so we have shown how to generate a sequence of steps that move $k+1$ discs.

The two parts above are sufficient to prove the general result. If one writes an abbreviation H_n for the statement "The Hanoi puzzle can be solved if there are n discs" then we proved H_1 directly, and then set up a chain $H_1 \Rightarrow H_2$, then $H_2 \Rightarrow H_3$, $H_3 \Rightarrow H_4$ and so on for ever. So you see that the inductive step in the proof is just a way of building up (all at once) a chain of simple "$\Rightarrow$" deductions.

It is **vital** in proofs by induction to have a base case. In many examples that you will come across it will be incredibly trivial, but it is still important to write it down.

In the above example the inductive step took the view that when proving stage n it was legal to assume that stage $n-1$ was true[2]. An alternative variant on the induction idea is that when proving stage n you assume that the result has already been proved for **all** values less than n, not just the special value $n-1$.

A lion may be asleep somewhere in a desert, can we find it or discover that it is absent?

Base case: If the desert has area less than 10 square metres then we can find the lion [since this example is slightly a joke I am going to assume that this remark is "obviously" true].

Inductive step: Divide the desert into two sub-deserts each of equal area. Call these N and S (or E and W if you like). Observe that their area is in each case smaller than that of the whole desert. Thus by an induction hypothesis we can find the lion (or its absence) in each sub-desert. If we find the lion in one part we report success, if it is not found in either sub-desert it is utterly absent.

This proof needs one more crucial component—an argument that if we take any desert at all and keep dividing it in two we will always end up with a region of sand that has area less than 10 square metres. The base case here is in fact a bit questionable, especially if you imagine a large desert always being split strictly North/South so that the 10 square metre base case is in fact a very very long but very very thin strip of territory. Such concerns do not damage the structure of the inductive proof, but should serve to warn you that all the fine details must be in place for a proof to be valid.

I will show one more example of a proof that uses a form of induction. The result I want to prove is that any arithmetic expression that uses just addition and multiplication

[2]Well, the notation that when proving stage $k+1$ I assumed the result true for k, but that amounts to the same thing!

and that has only even numbers written in it will evaluate to an even value. This will be structurally much closer to many of the proofs needed for DS&A methods. The inductive proof will be in terms of the number of operators in an expression.

Base case: With no operators at all the expression is just a number, and by the statement of the problem that is an even one.

Inductive step: If there is at least one operator then the whole expression is of the form $A \oplus B$ where $\oplus$ will be either an addition or multiplication operator, and A and B are sub-expressions. I will treat things like $2 + 4 + 6$ as $(2 + 4) + 6$ here. There are then two cases to consider

The leading operator is + Each of A and B will have fewer operators than the whole expression, hence by induction each will evaluate to an even number. The sum of two even numbers is even, hence the whole expression is even as required.

The leading operator is * Similarly, since the product of two even numbers is even.

4 Sets, Relations

Very many computer data structures are best reasoned about using things called "sets". For the purpose of this course a set is collection of things, and is written by listing the members of the set inside curly brackets.. So for instance the set whose members are the first five whole numbers might be written {0,1,2,3,4}. It is perfectly possible to have a set with no members at all, and it is then (obviously) written as {}, and known as the empty set. There are a number of rules about sets and a collection of things you are allowed to do with them:

- The members of a set should not be thought of as being there in an particular order, even though when you write down a representation of a set you have to list them somehow. So for instance {1,2,3}, {3,1,2} and {2,1,3} are all just different ways of writing down the same set.

- Any particular object can either be in a set or not. The consequence of this is that when you list the members of a set you should never see any duplicates. It is not possible for an object to be a member of a set twice or more times, so {1,1,1,2,2,3} is not a valid things to write.

- As well as having the null set {} it is (of course) quite proper to have sets that have a single member. Set members can be arbitrary things, including other sets. Thus we can have the number 1, and the sets {1}, {{1}}, {{{1}}}. These are all quite different things. For instance the set {{}, {{}}} is a set with two members (one of which is the empty set, the other is a set whose sole member is the empty set).

- It is quite common to use upper case letters to stand for sets and lower case ones for the items that may be members of those sets. The notation $x \in A$ is used to indicate set membership. x is an object and A is a set. Eg $3 \in \{1, 2, 3, 4, 5\}$ is true. The notation $A \subset B$ is used to indicate that A is a subset of B, ie it has as its members some selection from the members of B. Eg $\{2, 4\} \subset \{1, 2, 3, 4, 5\}$.
- The notation $A \cap B$ is a set whose members are just those things that are present in both A and B (the *intersection*), while $A \cup B$ has all the members that are in either (or both) A and B (the *union*).
- The exact rules for the treatment of very infinite sets are not needed in this course, but informal notation will be used to describe simple cases, such as the integers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$. The expression $\{x^2 \mid x \in \{0, 1, \ldots\}\}$ stands for $\{0, 1, 4, 9, \ldots\}$.
- Given a finite set you can find out how many members it has. The number is known as the cardinality of the set (or more simply as its size), and is sometime written $|A|$ as in $|\{2, 4, 6\}| = 3$.

An *ordered pair* is just a pair of things grouped together where (unlike the situation with sets) the ordering of the two items does match. Such pairs will be written with parentheses as in (1,2) rather than curly brackets. This idea will be extended to ordered triples, quadruples, 5-tuples and so on, and in general n-tuples. The objects in an n-tuple do not need to be all of the same sort. Eg here is a 4-tuple

```
(1, "string", {{}, {22}}, x}
```

where the third member of the 4-tuple is a set.

Given one or more sets there are ways of constructing bigger sets out of them. The cartesian product of two sets A and B is the set of ordered pairs (a, b) with $a \in A$ and $b \in B$. Eg $\{a, b\} \times \{1, 2\} = \{(a, 1), (a, 2), (b, 1), (b, 2)\}$. The powerset of a set A is the collection of all subsets of A. Eg. powerset($\{p, q, r\}$)= $\{\{\}, \{p\}, \{q\}, \{r\}, \{p, q\}, \{q, r\}, \{p, r\}, \{p, q, r\}\}$

You might like to convince yourself that $|A \times B| = |A||B|$ and $|\text{powerset}(A)| = 2^{|A|}$. For the powerset example try induction on the size of the set.

Given a set X , a *relation* is some property that may or may not hold between one member of X and another. For instance if attention is restricted to sets of numbers then the operator "<" for "is less than" is a valid relation, as would be "=" for equality. Sometimes people will want to use a general name, say R for an unspecified relation, and then rather than something concrete like $x = y$ they will write $x\mathrm{R}y$ to show that x relates to y under R.

Relations are not only things that arise with numbers. In a family tree (or many computer data structures you will come across later) the relation "is an ancestor of" can be relevant. Given a set of people one could specify the relationship "likes" (useful for a computer-based system for arranging seating plans at large dinners?). A final example is the "is married to" relation.

Relations can have different properties, and the examples given above can illustrate some of the important ones:

Reflexive: Given any member of the set, x say, does the relation hold true between x and itself. For $=$ it does, for $<$ and "is married to" it does not and for "likes" the status is unclear to me. Relations that have this property are known as reflexive. From any (possibly non-reflexive) relation you can derive something called the reflexive closure by forcing each item x to relate to itself but otherwise leaving conditions unchanged.

Symmetric: Of the examples given here, $=$ and "is married to" have the property that if (x,y) relate then (y,x) do too. This makes them symmetric, while $<$ is clearly not. Again the extent to which "likes" is a symmetric relation on any particular set is an interesting social consideration. The symmetric closure of a relation extends a relation to force symmetry. Eg the symmetric closure of $<$ is $\neq$.

Transitive: If $x = y$ and $y = z$ then we may deduce that $x = z$. Similarly for $<$, and these relations are, on account of this, known as transitive. There is such a thing as a transitive closure, which is discussed in the next section.

5 Relations and Graphs and Matrices

Relations can seem rather abstract things, of dubious utility. One of the things that makes them come alive in computer science is just an alternative way of looking at them (and especially at relations on finite sets). Take a set X and a relation R on it, and identify the members of X with nice dots drawn somewhere on a piece a paper. Then take the relation, and if two members in the set, say (x,y) are related (ie $x\mathrm{R}y$) draw a directed arc from the spot that stands for x to the one that stands for y. By a "directed arc" I mean that the line drawn has an arrow on it showing which way it goes, so that there is no possible confusion between the arc (x,y) and the one (y,x). The effect is that the relation has been represented as a graph.

Now set up a square table, with one row for each possible x or y and one column for each. Fill in the cell at position (x,y) with a true or false marker that indicates whether $x\mathrm{R}y$ holds. The relation has been represented as a matrix. And in passing we have shown that any graph can be represented as a matrix, and any matrix that has just boolean values can be interpreted as a graph. To give yourself concrete example, try drawing the graphs and matrices for the relations $=$ and $<$ as they apply to the set $\{1, 2, 3, 4, 5\}$.

Now I can come back to the transitive closure of a relation. A typical application is to start with a set consisting of cities, and a relation which is true if there is a direct non-stop rail link between the two cities involved. Then the transitive closure of this relation will indicate whether there is any way of travelling by rail between two places, ignoring the original requirements that the journey be direct and non-stop. In terms of graph operations this is now probably reasonably easy to visualise.

Interpreted in this new image, a reflexive closure just adds little loops to each vertex in the graph so you can do a small round trip and get back to where you started. Note the difference between being somewhere and being able to get somewhere by taking a single step of a journey—adding the loops does make a real difference.

A symmetric closure extends the rail network so that if it is possible to go from A to B then it is also possible to get back from B to A.

There are a great many natural and important problems that are naturally thought of in terms of graphs—and so relations can provide some mathematical notation and underpinning while sometimes boolean matrices may be a useful concrete representation for computers to use. Sample problems include:

1. Is the graph connected (ie each vertex can be reached from any other)? If not, how many pieces does it fall into?
2. What is the longest path you can take through the graph without visiting any vertex more than once. What is the longest path that does not traverse any edge more than once?
3. Given a connected graph, is there any vertex which if removed would leave it not connected? This is important for communication networks, in that such a vertex would be critical for the reliability of the whole net.
4. How many colours are needed to colour each vertex of the graph so that vertices that are joined by an edge have different colours?
5. Within the graph, where is the largest subset of vertices that are all mutually directly connected.
6. Given two graphs are they really the same shape, only differing in the way they happen to have been described?

A special sort of graph (and hence relation) has all arcs starting in one subset of its vertices (call that subset A), and ending in another (B), and only one arc issuing from any one vertex. This can be seen as a way of representing a function from the set A to B.

Looking at relations and functions as graphs is probably the easiest way of working out how many of them there are. For instance for a set X with size n the number of relations possible is 2^{n^2}. These range from the vacuuous one where the relation is never true to the almost equally silly one where it is always satisfied. See this by observing that each relation on X can be seen as an n by n matrix with boolean entries, so there are n^2 entries in all, and each can be either true or false (2 possible values) so there are 2^{n^2} possibilities in all.

6 Big-O and Θ notation

All the while in Computer Science we are concerned with how long things are going to take. It is almost always necessary to make a few simplifying assumptions before starting of cost estimation, and for algorithms the ones most commonly used are:

1. We only worry about the worst possible amount of time that some activity could take. The fact that sometimes our problems get solved a lot faster than that is nice, but the worst case is the one that is most important to worry about.

2. We do not know what brand of computer we are using, so rather than measuring absolute computing times we will look at rates of growth as our computer is used to solve larger and larger problems of the same sort. Often there will be a single simple number that can be used to characterise the size of a problem, and the idea is to express computing times as functions of this parameter. If the parameter is called n and the growth rate is $f(n)$ then constant multipliers will be ignored, so $100000f(n)$ and $0.000001f(n)$ will both be considered equivalent to just $f(n)$.

3. Any finite number of exceptions to a cost estimate are unimportant so long as the estimate is valid for all large enough values of n.

4. We do not restrict ourselves to just reasonable values of n or apply any other reality checks. Cost estimation will be carried through as an abstract mathematical activity.

Despite the severity of all these limitations cost estimation for algorithms has proved very useful, and almost always the indications it gives relate closely to the practical behaviour people observe when they write and run programs.

The notations bit-O and Θ are used as short-hand for some of the above cautions.

A function $f(n)$ is said to be $O(g(n))$ if there are constants k and N such that $f(n) < kg(n)$ whenever $n > N$.

A function $f(n)$ is said to be $\Theta(g(n))$ if there are constants k_1, k_2 and N such that $k_1 g(n) < f(n) < k_2 g(n)$ whenever $n > N$.

Note that neither notation says anything about $f(n)$ being a computing time estimate, even though that will be a common use. Big-O just provides an upper bound to say that $f(n)$ is less than something, while Θ is much stronger, and indicates that eventually f and g agree within a constant factor. Here are a few examples that may help explain:

$$\begin{array}{rcl}
sin(n) & = & O(1) \\
sin(n) & \neq & \Theta(1) \\
200 + sin(n) & = & \Theta(1) \\
123456n + 654321 & = & \Theta(n) \\
2n - 7 & = & O(17n^2) \\
\log(n) & = & O(n) \\
n^{100} & = & O(2^n) \\
1 + 100/n & = & \Theta(1)
\end{array}$$

Various important computer procedures have costs that grow as $O(n\log(n))$. In the proofs of this the logarithm will often come out as ones to base 2, but observe that $\log_2(n) = \Theta(\log_{10}(n))$ [indeed a stronger statement could be made—the ratio between them is utterly fixed], so with Big-O or Θ notation there is no need to specify the base of logarithms—all versions are equally valid.

7 Recurrence Formulae

When analysing algorithms one will often end up with a proof by induction that shows that the method described does indeed always solve the problem it was supposed to. Quite frequently this proof can be extended to yield a way of estimating the costs involved. Look back to the Tower of Hanoi example, and now we know that given a tower of n discs it can be moved from one peg to another, consider how many elementary moves will be used if we follow the recipe implicit in the inductive proof. To do this start by introducing a name for the cost, say $M(n)$ for the number of steps to move n discs. Then from the base case of the induction we have $M(1) = 1$. The induction step shows that success is possible by a route which gives

$$M(n) = M(n-1) + 1 + M(n-1) = 2M(n-1) + 1$$

This is a recurrence formula that we would like to solve to find some explicit representation of the cost growth function $M(n)$. Note that the proof we have does not show that this will necessarily be the most efficient way of moving the discs, just that it is one way that achieves the desired final configuration. Thus any result we get out from solving the recurrence will probably be put inside a Big-O to indicate that it is just an upper bound for the cost of solving the problem.

This course will not have either the time or inclination to show you all the clever ways there are of solving recurrence formula, and instead just provides a cook-book listing some of the more commonly arising ones and indicating their solutions. Symbols with names like k stand for constants, and will sometimes need to have values larger than 0 or 1 for the results quoted to be valid.

$$\begin{array}{rcl}
f(n) = f(n-1) + k & : & f(n) = \Theta(nk) \\
f(n) = k_1 f(n-1) + k_2 & : & f(n) = \Theta(k_1^n) \\
f(n) = k_1 f(n/k_1) + k_1 n & : & f(n) = \Theta(n \log(n)) \\
f(n) = f(n/k_1) + k_2 n & : & f(n) = \Theta(\log(n)) \\
f(n) = k_1 f(n/k_2) + \ldots & : & f(n) = \Theta(n^{\log(k_1/k_2)}) \\
f(n) = f(n-1) + f(n-2) & : & f(n) = \Theta(\phi^n)
\end{array}$$

where $\phi = (\sqrt{5}+1)/2 \approx 1.618034$, the golden ratio. In each case more careful analysis would specify the exact constraints on the values of the constants permitted, and limitations on the initial values of $f(0)$ or $f(1)$. Often an exact solution (not just one correct to within the constant factor that Θ permits) can be found, for instance for the Hanoi problem the solution is $M(n) = 2^n - 1$.

Appendix: Mathematical symbol glossary

This page illustrates the most common mathematical symbols you are likely to encounter. If any course uses a notation not on this page you should expect it to have been previously defined in that course. Otherwise you should feel no hesitation in requesting an explanation (or to ask me to include it in this glossary!).

$\alpha, \beta, \gamma, \delta$	greek letters
$\mathrm{A}, \mathrm{B}, \Gamma, \Delta$	upper case greek letters
0, 1, -2, 3.14159	numbers
$+, -, \times, /$	arithmetic operators
$<, >, \geq, =$	relational operators
$\wedge, \vee, \neg, \Rightarrow, \Leftrightarrow$	logical operators
$\forall, \exists$	logical quantifiers ('for all', 'there exists')
$\{\}$, $\{3,5,7\}$, $\{x \mid x^2 - 3x + 2 = 0\}$	sets
$\cap, \cup, \times, \rightarrow$	set operators (intersection, union, cartesian product, function space)
$\subseteq, \supseteq, =$	set relational operators
$\in$	set membership
$\|S\|$	number of members in set S
$\mathcal{P}(S)$	power set of S (set of all subsets)
$\mathbb{N}, \mathbf{Z}$	natural numbers (0,1,2,...), integers
$\lambda x.x + 1$	anonymous function.

Sometimes, as for induction, $A \wedge B \Rightarrow C$ is written (but beware this as a sloppy explanation):

$$\frac{A \qquad B}{C}$$

Examples (some of these are definitions of the corresponding symbols):

$$\begin{aligned}
x > 5 &\Rightarrow x^2 > 16 \\
x^2 \geq 9 &\Leftrightarrow (x \geq 3 \vee x \leq -3) \\
S \cap T &= \{x \mid x \in S \wedge x \in T\} \\
S \cup T &= \{x \mid x \in S \vee x \in T\} \\
S \times T &= \{(x,y) \mid x \in S \wedge y \in T\} \\
x \in S &\Rightarrow x \in S \cup T \\
n \in \mathbb{N} &\Rightarrow n + 1 \in \mathbb{N} \\
(\lambda x.x + 1)5 &= 6
\end{aligned}$$

Mathematical induction can be summarised as:

$$\frac{P(0) \qquad\qquad P(k) \Rightarrow P(k+1)}{(\forall n \in \mathbb{N}) P(n)}$$