Some Useful Information

1. $\sum_{i=m}^{n} cf(i) = c\sum_{i=m}^{n} f(i)$

2. $\sum_{i=m}^{n} (f(i) + g(i)) = \sum_{i=m}^{n} f(i) + \sum_{i=m}^{n} g(i)$

3. $\sum_{i=m}^{n} f(i) = \sum_{i=0}^{n} f(i) - \sum_{i=0}^{m-1} f(i)$ $\quad 0 \le m \le n$

4. $\sum_{i=m}^{n} c = c\sum_{i=m}^{n} 1 = c(n-m+1)$ $\quad m \le n$, c a constant

5. $\sum_{i=0}^{n} i^0 = \sum_{i=0}^{n} 1 = n+1$

6. $\sum_{i=0}^{n} i^1 = \sum_{i=0}^{n} i = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$

7. $\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}$

8. $\sum_{i=0}^{n} i^k = \frac{n^{k+1}}{k+1}$ + {lower powers of n}

9. $\sum_{i=0}^{n} c^i = \frac{c^{n+1}-1}{c-1}$ $\quad$ if $c \neq 1$
   $\quad\quad\quad\quad\quad\quad\quad\quad$ if $c = 1$, use (5)

10. $\sum_{i=0}^{n} \frac{1}{c^i} = \frac{c - \frac{1}{c^n}}{c-1}$ $\quad c \neq 0, 1$

(note: this is really the same as (9) with c replaced by 1/c.

11. $\sum_{i=1}^{n} \log(i) = \log(n!) \equiv n\log(n)$

remember: $n! = 1 \cdot 2 \cdot 3 \cdot \cdots \cdot n$
log is to any base
$\log(a \cdot b) = \log(a) + \log(b)$
$\log(a/b) = \log(a) - \log(b)$
$\log_a(a) = 1$
$\log(1) = 0$

RESERVE

12. $\sum_{i=1}^{n} \frac{1}{i}$ is called the "harmonic series" and is often denoted by $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$. It has no known closed form solution but $H_n \equiv \log(n)$ for large n. In terms of log base 2, it can be shown $\frac{\lfloor \log_2 n \rfloor + 1}{2} < H_n \le \lfloor \log_2 n \rfloor + 1$ In terms of natural logarithms: $\ln(n) < H_n < \ln(n) + 1$.

13. The number of ways to select m objects from a set of n objects, with replacement, is denoted by C_m^n or $\binom{n}{m}$ and has the value

$$\frac{n!}{m!(n-m)!}$$ remember $0! \equiv 1$.

14. $\binom{n}{m} = \binom{n}{n-m}$

15. $\sum_{i=0}^{n}\binom{n}{i} = 2^n$

16. if A is a set of n objects, the "power set of A" is denoted p(A) and is the collection of all subsets of A. Then p(A) has 2^n elements.
notation: if A has n elements then we write $|A| = n$.

17. $\sum_{i=1}^{n}\sum_{j=1}^{m} f(i,j) = \sum_{j=1}^{m}\sum_{i=1}^{n} f(i,j)$

18.

$$\sum_{i=1}^{n}\sum_{j=1}^{m} f(i)\cdot f(j) = \sum_{i=1}^{n}[f(i)\sum_{j=1}^{m} f(j)]$$

$$= [\sum_{j=1}^{m} f(j)]\cdot[\sum_{i=1}^{m} f(i)]$$

An "axiom" is an accepted true statement; i.e. a statement that needs no proof of validity.

A "lemma" is usually a statement that needs a proof of its validity, often though, it is rather simple to prove and its usefulness is usually that it is used as a step in the proof of a more complex statement.

A "theorem" is the more complex thing you are trying to prove. Its proof is usually the thing you are trying to solve or bears directly on the problem you are trying to solve.

A "corollary" is usually a straight forward consequence of a theorem, possibly a special case, and is of some interest in itself but may not really be related to the problem you are trying to solve.

Note: these definitions are not definitive in the sense that one person's corollary might be another person's theorem, etc.

Also note that definitions are often thought of as axioms.

Common "Proof" Techniques

The object is to "prove" the validity of some statement S. (sometimes you may want to "disprove" the validity of S, but that is the same as "proving" the validity of not S.) A proof that statement S is true consists of using a set of axioms, previously proven lemmas, theorems and/or corollaries, and the rules of logic to arrive at a conclusion that S is unequivocally and undeniably true. A "proof" is invalid (really, a contradiction in terms, since an invalid proof is not a proof) when either (1) an assumed axiom is not really an axiom, (2) an assumed theorem is not really true (i.e., its "proof" is really invalid), or (3) a rule of logic was improperly used or applied in the steps leading to the conclusion.

To prove S is true we are really trying to prove the statement

"if A then S"

is true, where A is a set of all relevant axioms and previously proven Theorems (and Lemmas and Corollaries). The validity of this statement is equivalent to the validity of S. The trick is to find the statements needed in the set A and the sequence of logic rules.

There are several ways to go about doing this. Three of the most common and most useful are (1) Direct Proofs, (2) Contradiction Proofs, and (3) Induction Proofs.

(1) Direct Proofs.

The idea here is to start with known definitions, axioms and proven theorems and then to try to prove a sequence of new results (lemmas) which lead closer and closer to Statement S. And finally you have built enough "tools" to draw the conclusion that S must be true. For example

If A then L_1
If $A \cup L_1$ then L_2
if $A \cup L_1 \cup L_2$ then L_3
. . .
If $A \cup L_1 \cup ... \cup L_k$ then S

The trick is to determine what each L_i statement should be and to prove the corresponding " If $A \cup L_1 \cup ... \cup L_{i-1}$ then L_i " statement.

(Notice: this is a classic case of Divide-and-Conquer applied to proving theorems)

(2) Contradiction Proofs

To prove "If A then S" true is equivalent to proving "If not S then not A" is true. Notice — this is not proving "not A" is true. This is proving the statement "if not S then not A" is true. When this statement is proven then, since we know not A is always false (since A is always true), we can conclude that not S must always be false, i.e., S must always be true.

So, how do we go about proving "if not S then not A"? We do it by assuming not S is a previously proven theorem and then by, e.g. the techniques in (1), try to conclude some statement in A must be false.

In essence the techniques used in a direct proof and a contradiction proof are identical, and usually it can be shown that if there is a direct proof there is also a contradiction proof and vice-versa. So, why use one over the other? Sometimes one way just seems to be clearer, easier or more straightforward than the other. The proofs of some theorems just seem to lend themselves to contradiction rather than a direct approach (and vice-versa). Sometimes it's simply a personal preference of the theorem prover.

(3) Induction Proofs

Proving the validity of a statement S by induction is probably the most misunderstood of all the proof techniques, yet it is also probably the easiest and most useful of the proof techniques for computer science. There are several variations and versions of the induction process. What I'll describe here is the most common.

Induction is not applicable in all cases. Usually it is used to establish the validity of statements S where S claims some property holds for some function f, where f is a function of an integer variable n, e.g. possibly

$$S: f(n) \geq c \text{ for all } n \geq 5.$$

where c is a constant, say. (C could also be variable and itself a function of n.)

The idea is the following and requires two steps.

(1) first you try to establish some relationship between f(i-1) and f(i) that holds for all integers i (at least for those ≥ 5). This might be given to you or it may be something you have to derive and prove holds. This relationship must be strong enough for you to make the following conclusion (which also must be proven.)

"If the property were to hold at i-1 then the relationship allows you to prove that it would also hold at i."

For example, with S as above

$$S: f(n) \geq c \text{ for all } n \geq 5$$

Suppose you are given, or can prove, that f(i-1) + 1 = f(i), (the relationship) for all integers i.

Now, "if the property were to hold at i-1," i.e. f(i-1) ≥ c, "then the relationship", i.e. f(i) = f(i-1) + 1, allows us to prove that it would also hold at i." i.e. f(i) ≥ c.

So, what we have proven is that if it holds at some i then it holds at all integers greater than i, since transitivity holds here, i.e.,

$$\text{if } f(i) \geq c \text{ then } f(i+1) \geq c,\ f(i+2) \geq c,\ \ldots .$$

There is one last step.

(2) You must show it does indeed hold for some integer i and determine what that i is. (in this case we want to make sure i is ≤ 5.) So what you try to do, usually, is to pick some value of i ≤ 5. Now try to prove, for that specific i, that f(i) ≥ c.

That's it.

Most discussions of induction give step (2) first and call it the "basis" step. In many ways it makes more sense to me to make it the second step.

An old analogy is proving you can climb to the top of ladder (assuming no loss of energy or fear of heights). You must prove two things:

(1) No matter where you are on the ladder, say rung i-1, you must prove you can make it up one more rung, rung i.

(2) You must prove you can get on the ladder in the first place.

Notice: these two steps are sufficient to prove you can get to the top. They both are also necessary: if you haven't proven (1) you don't know you can even get to the second step above where you got on, and if you haven't proven (2) you surely can't guarantee you can get to the top because you haven't guaranteed you can even get on the ladder.

Consider the following statements:

"For any simple graph G with p nodes and e edges we must have $e \leq \frac{p(p-1)}{2}$."

We may assume the nodes are labeled 1,2, ..., p. One axiom we have is a definition — a simple graph has at most one edge between any pair of nodes and no edge from a node back to itself.

A Direct Proof:

The "degree" of node i, d_i, is the number of edges incident to node i. By looking at each node in turn and counting the number of edges coming into each node, we will count $d_1 + d_2 + \cdots + d_p$ edges. Since each edge "comes into" exactly two different nodes, we will have counted each edge twice. Thus

$$2e = d_1 + d_2 + \cdots + d_p = \sum_{i=1}^{P} d_i$$

or

$$e = \tfrac{1}{2}\sum_{i=1}^{P} d_i.$$

Since no node can have more than one edge from each of the other p-1 nodes, we know $d_i \leq p-1$ for $1 \leq i \leq p$. Therefore

$$e = \tfrac{1}{2}\sum_{1}^{P} d_i \leq \tfrac{1}{2}\sum_{i-1}^{P} p-1 = \frac{P(P-1)}{2}$$

and we have proven the result. □

A Second Direct Proof:

There are p nodes and each edge can go between exactly two of them. So, the maximum number of ways you could place an edge in the graph is the number of ways you can select two nodes from a set of p nodes, i.e. $\binom{p}{2}$. Therefore

$$e \le \binom{p}{2} = \frac{p!}{2!(p-2)!} = \frac{p(p-1)}{2}.$$

And again we have proven the result. □

A Proof by Contradiction

We will prove the statement "if $e > \frac{p(p-1)}{2}$ then some axiom or known theorem must be false." We assume the nodes are labeled 1,2,...,P. For each edge make a copy of that edge. Label the first edge with the same label as one of the nodes it is adjacent to and label the second copy with the label of the other node. We now have $2e > p(p-1)$ edges each with a label on it. Notice that label i must occur exactly d_i times, thus $2e=d_1 + d_2 + \cdots + d_p$ (you've seen that before). Therefore

$$d_1 + d_2 + \cdots + d_p > p(p-1)$$

or

$$\frac{d_1 + d_2 + \cdots + d_p}{p} > p-1$$

The term on the left is the average degree of the p nodes. So this says the average degree is greater than p-1. But that cannot be true since we know the maximum degree is ≤ p-1.

Since our argument is true and we have found a contradiction with a known result, we can conclude that $e > \frac{p(p-1)}{2}$ must be false, i.e., $e \le \frac{p(p-1)}{2}$ must be true.

The theorem is again proven. □

Note this proof is very much like the "reverse" of our first direct proof. This possibility was mentioned earlier. Both are valid but here it seems the contradiction proof is a little more awkward and/or not as intuitive as the direct proof. (on other theorems the reverse may be true).

An Induction Proof.

(1) Let e_i be the maximum number of edge a graph with i nodes can have. When you allow one more node you can add at most one edge from the new node to each of the first i nodes. Therefore

$$e_{i+1} \le e_i + i$$

If the property were to hold for i, that is $e_i \le \frac{i(i-1)}{2}$, does the property hold for i+1? Notice that $e_i + i \le \frac{i(i-1)}{2} + i = \frac{i(i-1) + 2i}{2} = \frac{(i+1)i}{2}$. Thus $e_{i+1} \le \frac{(i+1)i}{2}$, and that's the property.

So if we can show, for a particular i, that $e_i \le \frac{i(i-1)}{2}$, the result will hold for all integers greater than that i.

(2) Consider i=1, i.e., G has 1 node and therefore $e_1 = 0$. Note $\frac{i(i-1)}{2}$ also equals zero. Therefore, 1 is our particular value.

The theorem is true for all graphs with one or more nodes. □

Constructive Induction

The goal, as before, is to obtain a closed form solution for a function given in a recursive form. Normal induction is a technique by which a 'quessed' closed form solution is verified. (Induction is not a method by which an answer is produced from scratch.) On the other hand, if one quess does not work out we can try another, etc. Hopefully the failure of one quess will lead to modifications that will provide a better next quess.

Constructive Induction is a technique in which we leave certain elements of the quess unspecified, that is, as unknown constants. Then, after the inductive argument is carried out, try to determine values for the constants which are consistent with the specification of the problem. Consider

$$f(n) = f(n-1)+n, \text{ and } f(1) = 1.$$

We know this to be $f(n) = n(n+1)/2$, in closed form, but suppose we didn't and only suspected f(n) was some quadratic function of n, i.e.,

$f(n) = an^2 +bn+c$, where a, b, and c are not yet known.

It is certainly possible to find values for a, b, and c so that $f(1) = 1$. So, suppose it is possible to find them so that

$$f(n-1) = a(n-1)^2 +b(n-1)+c, \text{ for some } n > 1.$$

Then $f(n) = f(n-1)+n = a(n-1)^2 +b(n-1)+c+n = an^2 +(b-2a+1)n-a-b+c$. The question now: is it possible to find a, b, and c so that

1) $a+b+c = 1$, and
2) $an^2 +(b-2a+1)n-a-b+c = an^2 +bn+c$?

The latter implies that we must have $b-2a+1 = b$ and $-a-b+c = c$, or that $a = 1/2$ and $a = b$. Therefore, with 1), we must have $c = 0$. The conclusion is then that

$$f(n) = n^2/2+n/2 = n(n+1)/2,$$

as we already knew, but were able to obtain by induction without actually utilizing that knowledge in the hypothesis. Notice, we do not need to still prove $f(n) = n(n+1)/2$ by standard induction.

Constructive induction has limitations. For instance, one needs to know at least the general form of the solution. Further, this form cannot be too complex or the mechanics of manipulating the induction hypothesis to the desired form becomes unmanageable (as it also would for standard induction.)

Notice that this technique is applicable when equality is replaced by inequality. This, and the comments in the last paragraph, might lead one to believe that constructive induction could be used in order analysis. Indeed that is the case since then our goal is to use rather 'simple' expressions. For example, showing $f(n) <= cg(n)$, where c is a constant and g(n) is often of the form n^k, $\log(n)$, $n\log(n)$, etc. To be even more specific, suppose we simply wanted to prove $f(n) = f(n-1)+n$ is order n^2. Then we merely need to show there exists a constant c, for large enough n, so that $f(n) \leq cn^2$. Letting the induction hypothesis be that $f(n-1) \leq c(n-1)^2$, we have

$$f(n) = f(n-1)+n \leq c(n-1)^2+n = c(n^2-2n+1)+n = cn^2-2cn+c+n,$$

or

$$f(n) \leq cn^2-(2c-1)n+c.$$

The right hand quantity is bound above by cn^2 for all $n \geq 1$ and $c \geq 1$. All that remains is to select c to satisfy the basis of $f(1) = 1$. Letting $c = \max\{1,f(1)\}$ is sufficient.

Other examples are admittedly not as straightforward. For instance, consider

$$f(n) = 4f(n/2)+n, \text{ for } n > 1 \text{ and } f(1) = 1.$$

By rather simple techniques, it is possible to show f(n) is order n^2. But constructive induction, when blindly applied, may be misleading. Begin with the normal hypothesis that $f(n/2) \leq c(n/2)^2$, for some constant c. Then we have

$$f(n) = 4f(n/2)+n \leq 4c(n/2)^2+n = cn^2+n.$$

Clearly, $cn^2+n > cn^2$, for all $n > 0$ and any positive constant c. Notice, this does not say $f(n)>cn^2$, only that the right hand side is larger than cn^2. So our desired conclusion may still be valid. If it is this only implies our hypothesis is simply not 'strong enough.' On the other hand it may be false — you just can't tell at this point.

> [note: a common fallacy here is to take the right hand side, cn^2+n, and conclude that $cn^2+n \leq c'n^2$ for some constant c'. Thus, that $f(n) \leq c'n^2$ and is therefore order n^2. This is in error. We MUST use the same constant c throughout. Otherwise it would be possible to show things like
>
> $$f(n) = f(n-1)+n <= cn,$$
>
> which we know to be n(n+1)/2.]

The remedy here is indicative of similar situations and lies in the statement above that our hypothesis was not strong enough, Here, not strong enough means not restrictive enough. Towards that idea, let our constructive induction hypothesis be that

$$f(n/2) \leq c(n/2)^2-b(n/2), \text{ for positive constants c AND b.}$$

Then,

$$f(n) = 4f(n/2)+n \leq 4(c(n/2)^2-b(n/2))+n = cn^2-2bn+n.$$

It is clear that this right hand side can be bounded above by cn^2-bn for any positive constant c and constant $b \geq 1$. Now, from the base condition, f(1), we must have

$$f(1) = 1 \leq c(1)^2-b(1) = c-b.$$

So we may choose b = 1 and c = 2, concluding that

$$f(n) \leq 2n^2-n.$$

Notice that a standard induction argument would now show this to be a valid conclusion, but, as mentioned before, this is unnecessary.

LOWER BOUND THEORY
Searching ordered lists with Comparison–Based Algorithms

Comparison-Based Algorithms: Information can be gained only by comparing key–to–element, or element–to–element (in some problems).

Given: An integer n, a key, and an ordered list of n values.
Question: Is the key in the list and, if so, at what index?

We have an algorithm. We don't know what it is, or how it works. It accepts n, a key and a list on n values. That's it.

It MUST, though, work pretty much as follows:

1) It must calculate an index for the first compare based solely upon n since it has not yet compared the key against anything, i.e., it has not yet obtained any additional information. Notice, this means for a fixed value of n, the position of the first compare is fixed for all data sets (of size n).

2) The following is repeated until the key is found, or until it is determined that no location contains the key:
   The key is compared against the item at the specified index.
   a) If they are equal, the algorithm halts.
   b) If the key is less, then it incorporates this information and computes a new index.
   c) If the key is greater, then it incorporates this information and computes a new index

There are no rules about how this must be done. In fact we want to leave it wide open so that we are not eliminating any possible algorithm.

After the first compare, there are two possible second compare locations (indexes). Neither depends upon the key or any item in the list: Just upon the result of the first compare. Every second compare on every set of n items will be one of these two locations.

Every third compare will be one of four locations. Every fourth compare will be one of eight locations. And, so on. In fact, we may look at an algorithm (for a given n) as being described by (or, possibly, describing) a binary tree in which the root corresponds to the first comparison, it's children to the possible second comparisons, their four children represent the possible third comparison, etc. This binary tree, called in this context a "decision tree," then depicts for this algorithm every possible path of comparisons that could be forced by any particular key and set of n values.

Observation 0: Every comparison–based search algorithm has it's own set of decision tree's (one for each value of n) – even if we don't know what or how it does its task, we know it has one for each n and are pretty much like the one described above.
Observation 1: For any decision tree and any root–leaf path, there is a set of date which will force the algorithm to take that path. The number of compares with a given data set (key and n values) is the number of nodes in the "forced" root–leaf path.
Observation 2: The longest root–leaf path is the "worst case" running time of the algorithm.
Observation 3. For any position i of $\{1, 2, \ldots, n\}$, some data set contains the key in that position. So every algorithm must have a compare for every index, that is, the decision tree must have at least one node for each position.

Therefore, all decision trees–for the search problem–must have at least n nodes in them.

All binary trees with n nodes have a root–leaf path with at least $\lceil \log_2(n+1) \rceil$ nodes (you can verify this by induction).

Thus, all decision trees defined by search algorithms on n items, have a path requiring $\lceil \log_2(n+1) \rceil$ compares.

Therefore, **the best any comparison–based search algorithm can hope to do** is $\log_2 n \approx \lceil \log_2(n+1) \rceil$.

This is the *comparison–based lower bound for the problem of searching an ordered list* of n items for a given key.

Comparison Based Sorting

Here, there is no "key." Typically, in comparison–based sorting, we will compare values in two locations and, depending upon which is greater, we might

1) do nothing,
2) exchange the two values,
3) move one of the values to a third position,
4) leave a reference (pointer) at one of the positions,
5) etc.

As with searching, above, each sorting algorithm has its own decision tree. Some differences occur: Leaf nodes are the locations which indicate "the list is now sorted." Internal nodes simply represent comparisons on the path to a leaf node.

As with searching, we will determine the minimum number of nodes any sorting algorithm (comparison–based) must have. Then, from that, the minimum height of all decision trees (for sorting) can be determined providing the proof that all comparison–based sorting algorithms must use at least this many comparisons.

Actually, it is quite simple now. Every decision tree starts with an unordered list and ends up at a leaf node with a sorted list. Suppose you have two lists, one is a rearrangement of the other. Then, in sorting them, something must be done differently to one of the lists (done at a different time). Otherwise, if the same actions are performed to both lists in exactly the same sequence, then one of them can not end up sorted. Therefore, they must go through different paths from the root of the decision tree. By the same reasoning, all n! different permutations of the integers 1, 2, . . ., n (these are valid things to sort, too, you know) must go through distinct paths. Notice that distinct paths end in distinct leaf nodes. Thus, there must be n! leaf nodes in every decision tree for sorting. That is, their height is at least $\log_2(n!)$. By a common result (one of Sterlings formula's) $\log_2(n!) \approx n\log_2(n)$.

Therefore, **all comparison–based sorting algorithms require $n\log_2(n)$ time.**