CS236 notes

Mark Wang

December 9, 2016

1.2 Simple Induction

Definition 1. Proof by **simple induction** is a method for proving statement

$$\forall n \in \mathbb{N}, P(n)$$

The method of induction consists of 2 steps
BASIS: Prove that $P(0)$ is true, ie. that predicate $P(n)$ holds for $n = 0$.
INDUCTION STEP: Prove that, for each $i \in \mathbb{N}$, if P(i) is true then $P(i+1)$ is also true.

Remark. The assumption that $P(i)$ holds in the indcution step of the proof is called the *induction hypothesis*. Bases case can be non-zero.

Example 1.1. For any $m, n \in \mathbb{N}$ such that $n \neq 0$, there are unique $q, r \in \mathbb{N}$ such that $m = q\dot{n} + r$ and $r < n$

Remark. Think about 2 cases. either $r < n - 1$ or $r = n - 1$

Example 1.2. We can use an unlimited supply of 4-cent and 7-cent postage stamps to make exactly any amount of postage that is 18 cents or more. Or that $\exists a, b \in \mathbb{N}, i = 4a + 7b$

Remark. Intuitively, try to juggle around value of a, b so that there is an excess of 1-cent, which satisfies for $i + 1$. In this case prove by cases to make it happen. Otherwise use proof by complete induction which is easier.

Definition 2. a is **divisible** by b if the division of a by b has no remainder.

$$b \mid a : \exists k \in \mathbb{N} : a = bk$$

Remark. Read $b \mid a$ as b divides a

Definition 3. An integer n is **prime** if $n \geq 2$ and the only positive integers that divide n are 1 and itself.

$$\{n \in \mathbb{N} : n \geq 2 \wedge m \mid n \Rightarrow m = 1 \vee m = n\}$$

Remark. Prime factorization of a natural number n is a sequence of primes whose product is n

1.3 Complete Induction

Definition 4. Proof by **complete induction** is a method for proving

$$\forall n \in \mathbb{N}, P(n)$$

BASIS: Prove that $P(n)$ holds for all $n \geq c$
INDUCTION STEP: Prove that, for each natural number $i > c$, if $P(j)$ holds for all natural numbers j such that $c \leq j < i$, then $P(i)$ holds as well.

Remark. It is important to ensure that both $j \geq c$ and $j < i$

Example 4.1. Any integer $n \geq 2$, has a prime factorization.

Proof. Define the predicate $P(n)$ as follows

$$P(n): \quad n \text{ has a prime factorization}$$

Use complete induction to prove that $P(n)$ holds for all integer $n \geq 2$. Let i be an arbitrary integer such taht $i \geq 2$. Assume that $P(j)$ holds for all integers j, such that $2 \leq j < i$. We muts prove that $P(i)$ holds as well. There are two cases
CASE 1: i is prime. Then $\langle i \rangle$ is a prime factorization of i. Thus $P(i)$ holds.
CASE 2: i is not prime. Thus there is a positive integer a that divides i such that $a \neq 1 \wedge a \neq i$. Let $b = {}^{i}/_{a}; i.e., i = a \cdot b$. Since $a \neq i \wedge a \lneq i$, it follows that a, b are both integers such that $2 \leq a, b \leq i$. Therefore, by the induction hypothesis, $P(a)$ and $P(b)$ both hold. That is, there is a prime factorization of a, say $\langle p_1, p_2, \ldots, p_k \rangle$, and there is a prime factorisation of b, say $\langle q_1, q_2, \ldots, q_l \rangle$. Since $i = a \cdot b$, it is obvious that concatenation of the prime factorisation of a and b, i.e. $\langle p_1, p_2, \ldots, p_k, q_1, q_2, \ldots, q_l \rangle$, is a prime factorisation of i. Therefore, $P(i)$ holds in this case as well. Therefore $P(n)$ holds for all $n \geq 2$

Remark. However, if we know the factorisation of all numbers less than i, then we can easily find a prime factorisation of i: if i is prime, then it is its own prime factorisation, and we are done; if i is not prime, then we can get a prime factorisation of i by concatenating the prime factorisations of two factors (which are smaller than i and therefore whose prime factorisation we know by induction hypothesis).

□

Example 4.2. Prove that ppostage of exactly n cents can be made using only 5-cents and 8-cents stamps

Proof. Define the predicate $P(n)$ as follows

$$P(n): \exists a, b \in \mathbb{N}, n = 5a + 8b$$

Use proof by complete induction to prove $P(n)$ holds for $n \geq 28$. Let i be an arbitrary integer such that $i \geq 28$, and assume that $P(j)$ holds for all j such that $28 \leq j < i$. We will prove that $P(i)$ holds as well.

CASE 1 or the BASIS: When $28 \leq i \leq 32$. We can make postage for all of them... Just have to calculate them...
CASE 2 or INDUCTION STEP: When $i \geq 32$. Let $j = i-5$ and therefore, by induction hypothesis, $P(j)$ holds. This means that $\exists a, b \in \mathbb{N}, j = 5a + 8b$.

$$\begin{aligned} i &= j + 5 \\ &= 5a + 8b + 5 \\ &= 5(a+1) + 8b && a_1 = a + 1, b_1 = b \\ &= 5a_1 + 8b_1 && a_1, b_1 \in \mathbb{N} \end{aligned}$$

Therefore, $P(i)$ holds as well. □

Remark. In this problem, a set of basis were discussed instead of one. This is to ensure that the choice of j satisfies $j \geq c$, which is required to use induction hypothesis.

Definition 5.

$$\lfloor x \rfloor = max\{m \in \mathbb{Z} : m \leq x\}$$
$$\lceil x \rceil = min\{m \in \mathbb{Z} : m \geq x\}$$

Definition 6. The Well Ordering Principle
Any nonempty set A of $\mathbb{N}$ contains a minimum element; ie, for any $A \subseteq \mathbb{N}$ such that $A \neq \emptyset$

$$\exists a \in A, \forall a' \in A, a \leq a'$$

4.1, 4.2 Structural Induction

Definition 7. Recursively Defined Sets
To define a set of objects

1. Define the simpliest or smallest objects in the set
2. Define ways in which larger more complex objects in the set can be constructed from smaller or simpler objects in the set

Definition 8. Let S be a set, A k-nary operator on S is a function

$$f : S^k \to S \qquad (\ S^k \text{ is the k-fold Cartesian product of S})$$

We say that $A \subseteq S$ is **closed** under f if,

$$\forall a_1, a_2, \ldots, a_k \in A, f(a_1), f(a_2), \ldots, f(a_k) \in A$$

Example 8.1. $\mathbb{N}$ of $\mathbb{Z}$ is closed under addition but not closed on subtraction

Definition 9. Principle of Set Definition by Recursion
Let S be a set, $B \subseteq S$, $m \in Z, m > 0$, and $f_1, f_2, \ldots, f_m$ be operators on S of arity $k_1, k_2, \ldots, k_m$, respectively,

$$S_i = \begin{cases} B, & \text{if } i = 0 \\ S_{i-1} \cup \bigcup_{j=1}^{m} \{f_j(a_1, a_2, \ldots, a_{k_j}) : a_1, a_2, \ldots, a_{k_j} \in S_{i-1}\} & \text{if } i > 0 \end{cases}$$

Then S_i is the smallest subset of S that contains B and is closed under $f_1, f_2, \ldots, f_m$

Definition 10. Proof by Structural Induction
To prove,

$$\forall x \in X : P(x) \qquad (\text{ where } X \text{ is defined recursively})$$

Basis: We prove that every smallest or simplest element of X is satisfied by P
Induction Step: We also prove that the infinitely many ways of constructing larger and more complex elements out of simpler and smaller ones *preserves* property P

3.1 Recursively Defined Function

Definition 11. Principle of function definition by recursion
Let $b \in \mathbb{Z}$, and $g : \mathbb{N} \times \mathbb{Z} \to \mathbb{Z}$ be a function. Then there is a unique function $f : \mathbb{N} \to \mathbb{Z}$ that satisfies the following:,

$$f(n) = \begin{cases} b & \text{if } n = 0 \\ g(n, f(n-1)) & \text{if } n > 0 \end{cases}$$

Note. Recursively defined functions may not be well defined,

1. lacking base case
2. inductive case either not defined for some n or form a loop over iteself

Example 11.1. The **The Fibonacci Function**

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$

which can be expressed as a closed-form formula,

$$F(n) = \frac{\phi^n - \hat{\phi}^n}{\sqrt{5}} \qquad (\phi = \frac{1+\sqrt{5}}{2}\ ,\ \hat{\phi} = \frac{1-\sqrt{5}}{2})$$

Definition 12. Principle of function definition by complete recursion:
Let k, l be positive integers, $b_0, b_1, \ldots, b_{k-1}$ be arbitrary integers, $h_1, h_2, \ldots, h_l : \mathbb{N} \to \mathbb{N}$ be functions such that $h_i(n) < n$ for each i, $1 \leq i \leq l$ and each $n \geq k$, and $g : \mathbb{N} \times \mathbb{Z}^l \to \mathbb{Z}$ be a function(Z^l denotes the l-fold Cartesian product of set $\mathbb{Z}$). Then there is a unique function $f : \mathbb{N} \to \mathbb{Z}$ that satisfies the following equation:

$$f(n) = \begin{cases} b_n & 0 \leq n < k \\ g(n, f(h_1(n)), f(h_2(n)), \ldots, f(h_l(n))) & n \geq k \end{cases}$$

2.2 Correctness Specification

1. A **precondition** for a program is an assertion involving some of the variables of the program;

2. A **postcondition** for a program is an assertion involving some of the variables of the program; this assertion states what must be true when the program ends in particular, it can describe what is a correct output for the given input.

3. A program is **correct** with respect to the specification (or the program meets the specification), if whenever the **precondition** holds before the program starts **execution**, then the program **terminates** and when it does, the **postcondition** holds. Note that implicitly stated is that all variable stated in pre- and post-condition cannot be altered.

2.3 Iterative Correctness of binary search

Definition 13. An **invariant** is a condition that can be relied upon to be true during execution of a program. A **loop invariant** is a condition that is true at the beginning and end of every execution of a loop. Consider a program containing a loop. Let P and Q be predicates of (some of) the variables of the program. We say that P is an **invariant** for the loop with respect to precondition Q if, assuming the programs variables satisfy Q before the loop starts, the programs variables also satisfy P before the loop starts as well as at the end of each iteration of the loop.

1. end of the 0-th iteration of the loop: the point in the program just before entering the loop

2. each iteration of the loop: includes this 0-th iteration

3. *An invariant is true at the end of each iteration of the loop, without explicitly saying that it is true before the loop.*

Algorithm 1: Iterative Binary Search

```
1  Function IterativeBinSearch (A, x)
       Input: A is a sorted array of length at least 1
       Output: Return t such that 1 ≤ t < length(A) and A(t) = x, if such a t exists,
               otherwise return 0.
2      f := 1
3      l := length(A)
4      while f ≠ l do
5          m := (f + l)%2
6          if A[m] ≥ x then
7              l := m
8          else
9              f := m + 1
10     if A[f] = x then
11         return f
12     else
13         return 0
```

Proof for IterativeBinSearch correctness
Suppose A is a sorted array of length at least 1. BinSearch(A, x) terminates and returns t such that $1 \leq t \leq length(A)$ and $A[t] = x$, if such a t exists; otherwise BinSearch(A, x) terminates and returns 0. Same as,

1. **Partial Correctness** Suppose A is a sorted array of length at least 1. If IterativeBinSearch(A, x) terminates then, when it does, it returns t such that $1 \leq t \leq length(A)$ and $A[t] = x$, if such a t exists; otherwise it returns 0.

2. **Termination** Suppose A is a sorted array of length at least 1. IterativeBinSearch(A, x) terminates.

To prove partial correctness, we prove that if precondition holds before program starts then the following is true at the end of each iteration of the loop,

$$1 \leq f \leq l \leq length(A), \text{ and if } x \text{ is in } A \text{ then } x \text{ is in } A[f..l].$$

The loop ensures that the element x being sought, if it is anywhere at all in the array, then it is in the part of the array that lies between indices f (as a lower bound) and l (as an upper bound).
Simply, we prove that the above is actually an **invariant** for the loop in IterativeBinSearch with respect to that programs precondition. More precisely,

Lemma 13.1. Suppose the precondition of IterativeBinSearch holds before the program starts. For each $i \in \mathbb{N}$, if the loop of BinSearch(A, x) is executed at least i times, then $1 \leq f_i \leq l_i \leq length(A)$, and if x is in A, then x is in $A[f_i..l_i]$

Proof. Define predicate,
$P(i)$: if the loop is executed at least i times, then *(i)* $1 \leq f_i \leq l_i \leq length(A)$, and *(ii)* if x is in A, then x is in $A[f_i..l_i]$
Details in notes.... □

Then we can use the loop invariant and the loop exit condition ($f = l$) to obtain the postcondition
Also we need to prove that the loop terminates, specifically the loop terminates.

Theorem 0.1. *By the well ordering principle, every decreasing sequence of natural numbers is finite.*

To prove the **termination of a loop** we typically proceed as follows. We associate with each iteration i of the loop a number k_i, defined in terms of the values of the variables in the i-th iteration, with the properties that

1. each k_i is a natural number
2. the sequence $k0, k1, k2, \ldots$ is decreasing

Then the loop must terminate, for otherwise we would have an infinite decreasing sequence of natural numbers.
In IterativeBinSearch we can associate with each iteration of the loop the value of the quantity $l_i - f_i$. This choice reflects the intuition that the reason the loop of BinSearch terminates is that the range of the array into which the search for x has been confined gets smaller and smaller with each iteration. The fact that $l_i - f_i$ is a natural number follows immediately from the fact that l_i and f_i are natural numbers and, by previous lemma, $f_i \leq l_i$. It remains to show that the sequence $l_0 - f_0, l_1 - f_1, l_2 - f_2, \ldots$ is decreasing.

Lemma 13.2. For each $i \in \mathbb{N}$, if the loop is executed at least $i+1$ times then $l_{i+1} - f_{i+1} < l_i - f_i$.

Proof. Details in notes... □

Lemma 13.3. For any integers f, l such that $f < l$, $f \leq \lfloor \frac{f+l}{2} \rfloor < l$

Proof. Details in notes... □

Summary

1. Correctness proofs of iterative programs are typically divided into two parts: one proving **partial correctness** (i.e., that the program is correct assuming it terminates); and another proving **termination** (i.e., that the program does indeed terminate).

2. The proof of termination typically involves associating a *strictly decreasing* sequence of *natural numbers* with the iterations of the loop, and apply the well-ordering principle.

3. The proof of partial correctness of an iterative program is typically based on a **loop invariant**. Proving that a statement is a loop invariant involves induction. A trick to finding the proper looop invariant is to come up with one that, when used in conjunction with exit condition, implies postcondition

2.7 Proof of correctness of recursive programs

Algorithm 2: Recursive Binary Search

```
1  Function RecBinSearch (A, f, l, x)
       Input: 1 ≤ f ≤ l ≤ length(A) and A[f..l] is sorted.
       Output: Return t such that f ≤ t ≤ l and A[t] = x, if such a t exists; otherwise,
               return 0.
2      if f = l then
3          if A[f] = x then
4              return f
5          else
6              return 0
7      else
8          m := (f + l)%2
9          if A[m] ≥ x then
10             return RecBinSearch (A, f, m, x)
11         else
12             return RecBinSearch (A, m + 1, l, x)
```

Lemma 13.4. Suppose that f and l are integers such that $1 \leq f \leq l \leq length(A)$, and that $A[f..l]$ is sorted when RecBinSearch(A, f, l, x) is called. Then this call terminates and returns t such that $f \leq t \leq l$ and $A[t] = x$, if such a t exists; otherwise it returns 0.

Proof. **The induction will be on the length of this subarray**. Complete induction is necessary as recursive calls works on half of length of array. □

3.2 Divide-and-conquer recurrences

Inductively defined functions arise from analysis of many recursively defined functions. We can analyze **Divide and Conquer algorithms**, including binary search and merge sort, by first constructing a recurrence relation describing its time complexity and unwind the function in closed form.

1 Recursive binary search

Definition 14. Suppose that f and l are integers such that $1 \leq f \leq l \leq \ length(A)$, and that $A[f..l]$ is sorted when **RecBinSearch(**A**,** f**,** l**,** x**)** is called. Then this call terminates and return t such that $f \leq t \leq l$ and $A[t] = x$, if such a t exists; otherwise it returns a 0.

Lemma 14.1. Suppose that f and l are integers such that $1 \leq f \leq l \leq length(A)$, and that $A[f..l]$ is sorted when RecBinSearch(A, f, l, x) is called. Then this call terminates and returns t such that $f \leq t \leq l$ and $A[t] = x$, if such a t exists; otherwise it returns 0.

Proof. Use induction to prove correctness on the length of subarray A. Note $length(A[f..l]) = l - f + 1$. Define predicate,
$P(k)$: if f, l are integers such that $1 \leq f \leq l \leq length(A)$ and $length(A[f..l]) = k$, and $A[f..l]$ is sorted when RecBinSearch(A, f, l, x) is called, then this call terminates and returns some t such that $f \leq t \leq l$ and $A[t] = x$, if such a t exists; otherwise it returns 0.
Now prove $P(k)$ holds for all $k \geq 1$
Basis:
Let $i = 1$, let f, l be integers such that $1 \leq f \leq l \leq length(A)$ and $length(A) = 1$. This means subarray A has one element and $f = l$. *if-statement* is executed and therefore program terminates by return statement at line 4 or 6. Since there is only one element in $A[f..l]$, then if x is in A, it must be $A[f] = x$, so the program returns f in line 4; on the other hand x is not in $A[f..l]$, $A[f] \neq x$ so program returns 0 as expected in line 6. Either way, program returns the right value then $P(1)$ holds.
Inductive step:
Let i be an arbitrary integer such that $i > 1$. Let f, l be integers such that $1 \leq f \leq l \leq length(A)$ and $length(A[f..l]) = i$, and suppose that $A[f..l]$ is sorted when RecBinSearch(A, f, l, x) is called. Assume that $P(j)$ holds for all j such that $1 \leq j < i$. We must prove that $P(i)$ holds as well. Since $length(A[f..l]) = i$ and $i > 1$, it follows that $f < l$. Therefore the else branch in lines 712 is executed. Let $m = (f + l)\%2$. then

$$f \leq m < l$$

□

2 MergeSort

Definition 15. If f, l are integers such that $1 \leq f \leq l \leq length(A)$ and $length(A[f..l]) = k$ then **MergeSort(**A**,** f**,** l**)** terminates and, when it does, $A[f..l]$ is sorted and all other elements of A are unchanged.

Algorithm 3: Merge

```
1  Function Merge (A, f, m, l)
       Input: 1 ≤ f ≤ m ≤ l ≤ length(A) and A[f..m], A[m + 1..l] are sorted
       Output: A[f..l] has the same element as before invocation (loop invariant), in
               sorted order, and all other element of A are unchanged
2      i ← f
3      j ← m + 1
4      k ← f
5      while i ≤ f ∧ j ≤ f do        /* Merge subarray until one is exhausted */
6          if A[i] < A[j] then
7              aux[k] ← A[i]
8              i ← i + 1
9          else
10             aux[k] ← A[j]
11             j ← j + 1
12         k ← k + 1
13     if i > m then                 /* Determine bounds of unexhausted array */
14         low ← j
15         high ← l
16     else
17         low ← i
18         high ← m
19     for t ← low to high do              /* Copy unexhausted array to aux */
20         aux[k] ← A[t]
21         k ← k + 1
22     for t ← fto l do                          /* Copy aux back to A[f..l] */
23         A[t] ← aux[t]
```

Algorithm 4: Merge Sort

```
1  Function MergeSort (A, f, l)
       Input: 1 ≤ f ≤ l ≤ length(A)
       Output: A[f..l] has the same element as before invocation (loop invariant), in
               sorted order, and all other element of A are unchanged
2      if f = m then                 /* subarray of length 1 already sorted */
3          return                                                /* do nothing */
4      else
5          m ← (a + b)//2
6          MergeSort (A, f, m)                              /* sort first half */
7          MergeSort (A, m + 1, l)                         /* sort second half */
8          Merge (A, f, m, l)                          /* merge 2 sorted halves */
```

Lemma 15.1. A **sorted array** A satisfies,

$$A[i] \leq A[i+1] \qquad (\forall i \in \mathbb{N}, 1 \leq i < \ length(A))$$

Lemma 15.2.

$$\exists i, j \in \mathbb{N}, 1 \leq i \leq j \leq length(A)$$

$A[i..j]$ denotes the subarray of A between indeces i and j with $n = length(A) = j - i + 1$

Lemma 15.3. For any integer $n > 1$, $1 \leq \lfloor n/2 \rfloor \leq \lceil n/2 \rceil < n$

Lemma 15.4.

$$\lfloor \frac{n+1}{2} \rfloor = \lceil \frac{n}{2} \rceil \text{ for some } n \in \mathbb{Z}$$

Remark. Mergesort ideas:

1. If $n = 1$, then array is already sorted
2. Split A to halves A_1, A_2, and recursively sort each
3. Merge the now sorted subarray A_1, A_2 to a single sorted array

Definition 16. Define $T(n)$ to be the maximum number of steps executed by a call to MergeSort(A, f, l), where n is the size of the subarray being sorted, i.e., $n = l - f + 1$. Then function T describes the (worst case) **time complexity** of MergeSort(A, f, l) as a function of size of array sorted.

$$T(n) = \begin{cases} c, & \text{if } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + dn, & \text{if } n > 1 \end{cases}$$

Note that c is some constant for sorting array of length 1. And the merging algorithm takes time proportional to n, with some constant d. To simplify the problem, assume n is a power of 2

$$T(n) = \begin{cases} c, & \text{if } n = 1 \\ 2T(n/2) + dn, & \text{if } n > 1 \end{cases}$$

Find closed form formula for $T(n)$ by **repeated substitution**, where we unwind recursive definition of $T(n)$ by applying induction step of the definition to smaller and smaller argument of T until we discover a pattern. Let $n = 2^k$, given $k \leq \log_2 n$, the $n/2^k$ is also power of 2. Therefore able to use induction hypothesis. The following conjecture can be proven using induction.

$$
\begin{aligned}
T(n) &= 2T(2^{k-1}) + d2^k \\
&= 2(2T(2^{k-2}) + d2^{k-1}) + d2^k \\
&= 2^2T(2^{k-2}) + 2d2^k \\
&\vdots \\
&= 2^kT(1) + kd2^k \\
&= cn + dn\log_2 n && (\ k = \log_2 n \text{ and } T(1) = 1)
\end{aligned}
$$

The purpose of this expression is to prove $T(n) \in \mathcal{O}(n \log n)$ and $T(n) \in \Omega(n \log n)$,

$$
\begin{aligned}
&\exists \kappa \in \mathbb{R}^+ : \forall n \geq 2 : T(n) \leq \kappa n \log_2 n \\
&\exists \kappa \in \mathbb{R}^+ : \forall n \geq 2 : T(n) \geq \kappa n \log_2 n \\
&\text{where the value of } \kappa \text{ depends on } c, d
\end{aligned}
$$

Theorem 0.2. *Prove $\exists \kappa \in \mathbb{R}^+, \forall n \geq 2, T(n) \leq \kappa n \log_2 n$ where $T(n) = cn + dn \log_2 n$ if n is a power of 2.*

Proof.
Let $\hat{n} = 2^{\lceil log_2 n \rceil}$, here $\hat{n}$ is the smallest integer that is a power of 2 greater than or equal to n, i.e.,

$$\frac{\hat{n}}{2} < n \leq \hat{n}$$

$$
\begin{aligned}
T(n) &\leq T(\hat{n}) && \text{(here requires proving } T(n) \text{ non-decreasing by induction)} \\
&\leq c\hat{n} + d\hat{n}\log_2 \hat{n} && (\text{ by } T(n) = cn + dnlog_2 n \text{ if } n \text{ is a power of 2}) \\
&\leq 2cn + 2dn \log_2 2n && (2n > \hat{n}) \\
&= 2cn + 2dn + 2dn \log_2 n \\
&\leq 2cn \log_2 n + 2dn \log_2 n + 2dn \log_2 n && (\log_2 n > 1 \text{ for } n \geq 2) \\
&= \kappa n \log_2 n && (\text{let } \kappa = 2c + 4d;\ \kappa > 0)
\end{aligned}
$$

Therefore $\exists \kappa \geq 0, \forall n \geq 2, T(n) \leq \kappa n \log_2 n$ or that $T(n) \in \mathcal{O}(n \log_2 n)$ □

Definition 17. Big O notation: function $g(n)$ is in $O(f(n))$ iff

$$\exists c \in \mathbb{R}^+ : \exists N \in \mathbb{N} : \forall n \geq N \Rightarrow (g(n) \leq cf(n))$$

Definition 18. General form of divide-and-conquer algorithm
An **instance** is a legitimate input for the problem. A **solution** of an instance is an output for the instance. Each instance has a **size**.

Steps for solving a large problem,

1. Divide up the given instance of size n into a total of a smaller instances of the *same problem*, each of size *roughly* $\frac{n}{b}$. ($\frac{n}{b}$)
2. Recursively solve each of the smaller instances (valid because smaller they are of same problem)
3. Combine the solutions to the smaller instances into the solution of the given large instance.

Step 1 and 3, i.e. dividing up instances and combining solutions to the smaller instances, is given by polynomial time dn^l. Recursive solution of smaller instances require $a_1T(\lceil\frac{n}{b}\rceil)+a_2T(\lfloor\frac{n}{b}\rfloor)$ steps; this is time required to solve a_1 instances of size $\lceil\frac{n}{b}\rceil$ and a_2 instances of $\lfloor\frac{n}{b}\rfloor$. **Recurrence relationship** of $T(n)$ follows,

$$f(n)=\begin{cases}c, & \text{if } 1\leq n<b\\ a_1T(\lceil\frac{n}{b}\rceil)+a_2T(\lfloor\frac{n}{b}\rfloor)+dn^l, & \text{if } n\geq b\end{cases}$$

where $a_1,a_2\in\mathbb{N}, a=a_1+a_2$, $b\in\mathbb{N}, b>1$. As an example, MergeSort has $a_1=1$, $a_2=1$, $b=2$, $l=1$

To find the **closed form** of $T(n)$, consider simplify by considering n as power of b. Then,

$$f(n)\begin{cases}c, & \text{if } n=1\\ aT(\frac{n}{b})+dn^l, & \text{if } n>1\end{cases}$$

where $a=a_1+a_2$

Theorem 0.3. *There is a constant $\kappa\geq 0$ (that depends on a,b,c,d and l) so that, for all integers $n\geq b$ that are powers of b, the function $T(n)$ satisfies,*

$$f(n)=\begin{cases}\kappa n^l, & \text{if } a<b^l\\ \kappa n^l\log_b n, & \text{if } a=b^l\\ \kappa n^{\log_b a}, & \text{if } a>b^l\end{cases}$$

Remark. Correspondingly, we can expand the theorem to all of n and set up the upper and lower bounds of $f(n)$, thereby poving

$$T(n)\in\begin{cases}\Theta(n^l), & \text{if } a<b^l\\ \Theta(n^l\log_b n), & \text{if } a=b^l\\ \Theta(n^{\log_b a}), & \text{if } a>b^l\end{cases}$$

Definition 19. Closest Pair Problem Given n points in metric space, find a pair of point with closest distance between them. By brute force, the algorithm requires $O(n^2)$, It is however possible to achieve $O(n\log_2 n)$ with divide-and-conquer following

1. $O(1)$ Split set of points S of size n into roughly two subsets S_1, S_2 such that $|S_1| \approx |S_2|$ with line l based on x-coordinate
2. $T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor)$ Recursively compute mimimum distance m_1, m_2 for S_1, S_2.
3. $O(1)$ Set $d = Minimum(m_1, m_2)$
4. $O(n)$ Eliminate points that are over d from l
5. $O(\log n)$ Sort rest of the points based on y-coordinate
6. $O(n)$ Scan the remaining points in the y order and compute the distances of each point to its five neighbors. (This is because a rectangle of width d and height $2d$ can contain at most 6 points such that any 2 points are at distance d apart. Implies that there's at most 6 point at the other set such that their distance to each other is at least d)
7. update d if any of distances was less than d

7.1 Introduction to Formal Language

Terminology

1. **Alphabet**: a set Σ whose elements are called **symbols**, i.e. $\{0, 1\}$
2. **String**: a finite sequence of symbols from Σ, i.e. 01001. The empty string is denoted as ϵ. Note the strings are representatively concatenated.
   (a) **Length** of string $|x|$
   (b) **Concatenation** of string $x \circ y$ is xy
   (c) **Reversal** of string $(x)^R$ is obtained by listing elements of x in reverse order
   (d) **Exponentiation** k-th power of x is x^k, represented by

$$x^k = \begin{cases} \epsilon, & k = 0 \\ x^{k-1} \circ x, & x > 0 \end{cases}$$

   Exponentiation of strings is repeated concatenation.
   (e) **Equality**: Two strings are equal if $|x| = |y|$ and if every i-th item of x is same as i-th item of y
   (f) **Substring**: String x is a substring of y if exists x_1, x_2 such that $y = x_1 x x_2$. If $x_1, x_2 \neq \epsilon$, then x is a **proper substring** of y.
   (g) **Prefix/Suffix**: A string x is a prefix for y if exists x_1 such that $xx_1 = y$, if $x_1 \neq \epsilon$ then x is a proper substring of y. Similarly for Suffix.

3. The set of all string over alphabet Σ is denoted as Σ^*. Σ^* can also be defined recursively by construction...

$$\textbf{Basis}: \epsilon \in \Sigma^*$$
$$\textbf{Inductive step}: \text{If } x \in \Sigma^* \text{ and } a \in \Sigma, \text{ then } xa \in \Sigma^*$$

Theorem 0.4. *For all strings x and y, $(xy)^R = (x)^R(y)^R$.*

Theorem 0.5. *A (formal) **language** L over the set Σ is a subset of Σ^**

Remark. A language may be infinite, but each *string* in the language must be finite. Also $\emptyset$ and $\{\epsilon\}$ are different languages

Definition 20. Operations on Language: Let L, L' be languages over Σ

1. **Complementation:** $\overline{L} = \Sigma^* - L$
2. **Union:** $L \cup L' = \{x : x \in L \vee x \in L'\}$
3. **Intersection:** $L \cap L' = \{x : x \in L \wedge x \in L'\}$
4. **Concatenation:** $L \circ L' = \{xy : x \in L, y \in L'\}$. Note $\emptyset \circ L = L \circ \emptyset = \emptyset$
5. **Kleene star:** $L^\star$ of language L is the set of all possible concatenation of 0 or more strings in L, defined recursively as

$$\textbf{Basis:} \quad \epsilon \in L^\star$$
$$\textbf{Inductive step:} \quad \text{If } x \in \text{Ł}^\star \text{ and } y \in L, \text{ then } xy \in L^\star$$

   Therefore, a string x belongs to L if and only if either $x = \epsilon$ or there exists some integer $k \geq 1$ and strings $x_1, \ldots, x_k \in L$ such that $x = x_1 x_2 \ldots x_k$.

6. **Language Exponentiation:** for any $k \in \mathbb{N}$, L^k is the set of strings obtained by concatenating k strings of L. For any k,

$$L^k = \begin{cases} \{\epsilon\}, & k = 0 \\ L^{k-1} \circ L. & k > 0 \end{cases}$$

   Note $L^1 = L$

7. **Reversal:** The reversal of L, $Rev(L)$ is the set of reversals of the strings in L

$$Rev(L) = \{(x)^R : x \in L\}$$

7.2 Regular Expressions

Definition 21. Regular expressions is a notation for describing languages.
Let Σ be a finite alphabet, the set of **regular expression** RE (over Σ) is the smallest set such that

BASIS: $\emptyset, \epsilon, a$ (for each $a \in \Sigma$) belong to RE
INDUCTIVE STEP: If R and S belong to RE, then $(R + S)$, (RS), R^* also belong to RE

Use these words to describe the language denoted by the regular expression.

1. **alteration** $+$, informally meaning *or*
2. **concatenation**, informally meaning *followed by*
3. **repetition** $*$, informally meaning *zero or more repetitions of*

Definition 22. The **language denoted by a regular expression** R is defined by structural induction on R,
Basis: Either $R = \emptyset$, or $R = \epsilon$, or $R = a$, for some $a \in \Sigma$. For each case we define $L(R)$,

- $L(\emptyset) = \emptyset$ (the empty language consisting of no strings)
- $L(\epsilon) = \epsilon$ (the language consisting of just empty string)
- for any $a \in \Sigma$, $L(a) = \{a\}$ (langauge consisting of a one-symbol sting a)

Inductive step: Either $R = S + T$, $R = (ST)$, or $R = S^*$, for some expression S and T. We define $L(R)$,

- $L((S + T)) = L(S) \cup L(T)$
- $L((ST)) = L(S) \circ L(T)$
- $L(S^*) = (L(S))^*$

Remark. Note concatenation takes precedence over union. Also we might want to prove equivalence of a language and a language denoted by a regular expression, specifically,

$$L(R) = L$$

To prove $x \in L(R) \Rightarrow x \in L$, we unwind regular expression using definition of language listed previously and express x as a concatenation of strings.
To prove $x \in L \Rightarrow x \in L(R)$, we try to break down x according to requirement of the language and categorize them into corresponding langugaes, whose composition is the desired $L(R)$

Definition 23. Equivalence of Regular Expressions Two regular expressions are equivalent, $R \equiv S$, if they denote the same language, i.e. $L(R) = L(S)$

- $(R + S) \equiv (S + R)$
- $((R + S) + T) \equiv (R + (S + T))$
- $((RS)T) \equiv (R(ST))$
- $(R(S + T)) \equiv ((RS) + (RT))$
- $((S + T)R) \equiv (SR + TR)$
- $(R + \emptyset) \equiv R$
- $(R\epsilon) \equiv R \equiv (\epsilon R)$
- $(\emptyset R) \equiv \emptyset \equiv (R\emptyset)$
- $R^{*^*} \equiv R^*$
- $((\epsilon + R)R^*) \equiv R^*$

Remark. We can use these equivalent expressions to simplify regular expressions

7.3 Definitive Finite State Automaton

A Definitive Finite State Automaton is a mathematical model of a machine which, given any input string x, accepts or rejects x. The automaton has a finite set of states, including a designated initial state and a designated set of accepting states.
It is customary to represent automata as directed graphs, with nodes (circles) corresponding to states, and edges (arcs) labeled with the symbols of the alphabet. An edge from state q to state q', labeled with symbol a, indicates that if the current state is q and the current input symbol is a, the automaton will move to state q. The initial state of the automaton is indicated by drawing an unlabeled edge into that state coming from no other state. The accepting states are indicated by double circles.

Definition 24. A **DFSA** M is a quintuple $M = (Q, \Sigma, \delta, s, F)$

1. Q is a finite set of **states**.
2. Σ is a finite **alphabet**.
3. $\delta : Q \times \Sigma \to Q$ is the **transition function**. In terms of the diagrammatic representation of the FSA, $\delta(q, a) = q'$ means that there is an edge labeled a from state q to state q'.

4. $s \in Q$ is the start or **initial state**.

5. $F \subseteq Q$ is the **set of accepting states**.

Remark. we can define the **extended transition function** $\delta^* : Q \times \Sigma^* \to Q$. Intuitively, if $q \in Q$ and $x \in \Sigma$, $\delta(q, x)$ denotes the state in which the automaton will move after it processes input x starting in state q. If $\delta^*(q, x) = q'$, we say that x takes the automaton M from q to q'

Remark. DFSA is **complete** is there is a transition on every symbol from each state.

Definition 25. A string $x \in \Sigma^*$ is **accepted** by M if and only if $\delta^*(s, x) \in F$, i.e. if and only if x takes the automaton from the initial state to an accepting state.The **language accepted** by a DFSA M, denoted $L(M)$, is the set of all strings accepted by M.

Definition 26. Conventions for DFSA diagrams

- **Combining transitions**
- **Eliminating dead states**. A state q_{reject} is dead if no accepting state is reachable from q_{reject}, i.e. there is no path that starts at q_{reject} and ends at accepting state.

Definition 27. Designing and proving correctness of DFSA For example, A DFSA that accepts the language,

$$L_1 = \{x \in \{0, 1\}, x \text{ has odd number of 0s and odd number of 1s}\}$$

An **state invariant** for a particular state describes what is true about the state. To prove state invariant for one state, we need to use induction on length of input string x and prove state invariant for each and every states, i.e.,

$$P(x) : \delta^*(q_0, x) = \begin{cases} q_0, & \text{if } x \text{ has an even number of 0s and an even number of 1s} \\ q_1, & \text{if } x \text{ has odd number of 0s and an even number of 1s} \\ q_2, & \text{if } x \text{ has an even number of 0s and an odd number of 1s} \\ q_3, & \text{if } x \text{ has an odd number of 0s and odd number of 1s} \end{cases}$$

Note that only if is required for state invariant predicate $P(x)$.Then we prove $P(x)$ holds for all strings x by structural induction.

Proof.
BASIS: when $x = \epsilon$, x has zero(even) number of 0s and zero(even) number of 1s and also $\delta^*(q_0, \epsilon) = q_0$, $P(x)$ holds.
INDUCTIVE STEP: when $x = ya$, where $y \in \Sigma^*$ and $a \in \Sigma$. Assume that $P(y)$ holds, and discuss 2 cases where a can be any symbol in the alphabet. Essentially, we concatenate another character from Σ and see if the state invariant holds. □

Once we proved state invariant, we prove equivalence of language to show that a DFSA accepts a certain language

$$L = L(M) \iff L \subseteq L(M) \land L \supseteq L(M)$$

Remember a string is in $L(M)$ if $\delta^*(q_0, x) \in F$, the accepting states.

Definition 28. Systematically finding the union of DFSA M_1 and M_2, $M = M_1 \cup M_2$. If M_1 has $Q_1 = \{p_0, p_1\}$ and M_2 has $Q_2 = \{q_0, q_1\}$, then their union M has the same alphabet, and a different state $Q = Q_1 \times Q_2 = \{(p_0, q_0), (p_0, q_1), (p_1, q_0), (p_1, q_1)\}$, start state $s = (p_0, q_0)$, and final states including states where any of the states were accepted in M_1 or M_2

7.4 Nondeterministic finite state automata

In a DFSA, a given state and current input symbol uniquely determine the next state of the automaton. There is a variant of finite state automata, called **nondeterministic finite state automata**, abbreviated NFSA, where this is not the case: From a given state, when the automaton reads an input symbol a, there may be **several states to which it may go next**. Furthermore, the automaton may spontaneously move from one state to another without reading any input symbol; such state transitions are called **ϵ-transitions**. The computation of a NFSA on input x corresponds to not a single path, as in the case of DFSA, but to **a set of paths**.

the NFSA accepts x if there is (at least) one computation path that it could follow on input x that ends in an accepting state. A string is rejected only if *every* computation path the NFSA could have followed ends in a nonaccepting state.

Presentationally, a FSA is considered a NFSA if there are more than one edge with the same symbol originating from the same node.

Definition 29. nondeterministic finite state automaton (NFSA) is a quintuple $M = (Q, \Sigma, \delta, s, F)$, where

- Q is a finite set of **states**
- Σ is a finite alphabet
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to \rho(Q)$ is the **transition function** Note here $\rho(Q)$ represents proper subset of Q, a set.
- $s \in Q$ is the start or **initial** state
- $F \subseteq Q$ is the set of **accepting** states

Remark. Note **extended transition function**

$$\delta^* : Q \times \Sigma^* \to \rho(Q)$$

A NFSA $M = (Q, \Sigma, \delta, s, F)$ **accepts** a string $x \in \Sigma^*$ if and only if $\delta^*(s, x) \cap F \neq \emptyset$ (In other words M accepts x if and only if there is at least one of the possible states in which the automaton could be after processing input x is an accepting state) **The language accepted by** M, $L(M)$, is the set of strings accepted by M.

Theorem 0.6. ***Equivalence of DFSA and NFSA*** *Any language that is accepted by a NFSA is accepted by a DFSA*
DFSA is a special case of NFSA, where we restrict transition function in certain ways. Therefore, NFSA is at least as powerful as DFSA. We can construct a DFSA from a NFSA via ***subset construction***

Definition 30. Proof of correctness for NFSA Just think about the path to accepted states and express it in terms of language; also think about the language and extrapolate what path the string will take the automata to, and if the final state is one of the accepting states.

7.5 Closure properties of FSA-accepted languages

Theorem 0.7. *The class of languages accepted by FSA is closed under complementation, union, intersection, concatenation, and Kleene star operation. In other wards, if L and L' are langauges that are accepted by FSA, then so are all of the following $\bar{L}, L \cup L', L \cap L', L \circ L', L^*$*

7.6 Equivalence of regular expression and FSA

Theorem 0.8. *For every regular expression R there is a FSA M such that*

$$L(M) = L(R)$$

Remark. Intuitively we recursively construct FSA that accept the language by the subexpressions of R, and combine these automata with ϵ transition to accept the language denoted by the entire expression R.

- we use an ϵ transition to concat two strings
- we use branched ϵ transition to make unions
- we use an ϵ transition leading to a previous state to represent Kleene star.

7.6 Regular Languages

Definition 31. Let L be a language, then the following statements are equivalent,

1. L is denoted by a regular expression

2. L is accepted by a deterministic FSA

3. L is accepted by a nondeterminstic FSA

A language is called **regular** if and only if it is denoted by some regular expression, or equivalently, if and only if it is accepted by a FSA.

7.7 Proving nonregularity: the Pumping Lemma

There are languages not accepted by FSA because FSA has only a fixed number of states, implying that it can only remember a bounded amount of things. For example,

$$\{0^n 1^n : n \geq 0\}$$

The **Pumping lemma** can be used to prove that languages are not regular. Intuitively, it states that any sufficiently long string of a regular language L has a nonempty substring which can be repeated ('pumped') an arbitrary number of times, with the resulting string still being in L.

Theorem 0.9. ***Pumping Lemma*** *Let $L \subseteq \Sigma^*$ be a regular language. Then there is some $n \in \mathbb{N}$ (that depends on L) so that every $x \in L$ that has length n or more satisfies the following property*

$$\textit{There are } u, v, w \in \Sigma^* \textit{ such that } x = uvw,\ v \neq \epsilon, \textit{ and } uv^k w \in L, \textit{ for all } k \in \mathbb{N}$$

Remark. To prove that a language is not regular, we assume for contradiction that L is regular. Therefore we let $n \in \mathbb{N}$ be the natural numbe which the Pumping lemma assserts that exists if L is regular. And then construct a string $x = uv^k w$ that is not in the language for all $k \in \mathbb{N}$. The trick is trying to pick an exponent m such that $m > n$ so that pumping must have happened if L is regular.