Dream.in.Code Computer Science Cheat Sheet

Michael Levet

August 30, 2013

Boolean Logic

- Logical And: An expression $p \wedge q$ is true only when both p and q are true.
- Logical Or: An expression $p \vee q$ is true when p is true, q is true, or both are true.
- Logical Not: The logical not operation inverts the truth value. So if p is true, then $\neg p$ returns false. Similarly, if p is false, then $\neg p$ is true.
- Exclusive Or (xor): An expression $p \oplus q$ is only true when p is true or q is true, but not both.
- Implication: An implication $p \implies q$ reads "p implies q." It can be thought of as an if statement. If "p" (the sufficient condition) is true, then "q" (the necessary condition) is also true. Another way to think about it is that $p \implies q \equiv true$ only when q is true. $p \implies q \equiv \neg p \vee q$.
- Biconditional: A biconditional $p \leftrightarrow q$ reads "p if and only if (iff) q." A biconditional is a definiton, and can be thought of as two implications: $p \implies q$ and $q \implies p$.
- Inverse: Given $p \implies q$, the inverse is $\neg p \implies \neg q$. If the original implication is valid, then the inverse is not valid.
- Converse: Given $p \implies q$, the converse is $q \implies p$. The converse is not necessarily true, unless there is a biconditional (so $q \implies p$).
- Contrapositive: Given $p \implies q$, the contrapositive is $\neg q \implies \neg p$. Note that $p \implies q \equiv \neg q \implies \neg p$. So if the implication is true, so is its contrapositive.
- DeMorgan's Law: $\neg(p \wedge q) \equiv \neg p \vee \neg q$, and $\neg(p \vee q) \equiv \neg p \wedge \neg q$.
- Modes Ponens: $((p \implies q) \wedge p) \implies q$ reads "If p implies q, and p is known, q can be inferred."
- Modes Tollens: $((p \implies q) \wedge \neg q) \implies \neg p$ reads "If p implies q, and the conclusion (q) is false, then the premise (p) is false as well."
- Distribution: $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$, and $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$.
- Universal Bounds: $p \vee true \equiv true$ and $p \wedge false \equiv false$.
- Idempotence: $p \vee p \equiv p \wedge p \equiv p$.
- Identity: $p \vee false \equiv p$ and $p \wedge true \equiv p$.
- Absorption: $p \wedge (p \vee q) \equiv p$ and $p \vee (p \wedge q) \equiv p$.

First-Order Predicate Logic

- Existential Quantifier: Denoted $\exists x \in D$. This reads "there exists some x in D."

- Universal Quantifier: Denoted $\forall x \in D$. This reads "for all x in D."
- Negation of Existential Quantifier: If $\exists x \in D$ such that $p(x)$, then the negation is: $\forall x \in D, \neg p(x)$.
- Negation of Universal Quantifier: If $\forall x \in D, p(x)$, then the negation is: $\exists x \in D$ such that $\neg p(x)$.

Set Theory

- Containment: $x \in A$ is read that x is an element of the set A.
- Set Union: $X = A \cup B$ is defined such that $X = \{x : x \in A \vee x \in B\}$.
- Set Intersection: $X = A \cap B$ is defined such that $X = \{x : x \in A \wedge x \in B\}$.
- Universal Set: Set of all elements in a domain.
- Null Set: Contains no elements. Denoted $\emptyset$.
- Set Complement: Denoted A^C, the complement of A contains the elements as follows: $A = \{a : a \in U \wedge a \notin A\}$, where U is the universal set.
- Set Difference: Denoted $X = A - B = A \cap B^C$, $X = \{x : x \in A \wedge x \notin B\}$.
- Symmetric Difference: Denoted $A \triangle B$, which is equivalent to $(A - B) \cup (B - A)$.
- Cartesian Product: Denoted $X = A \times B$. This produces pairs of the form (a, b). So $X = \{(a, b) : a \in A \wedge b \in B\}$.
- Power Set: Denoted $P(X)$, the power set contains all possible subsets of X, including the null set. If X is finite, then $|P(X)| = 2^{|X|}$. As an example, let $X = \{1, 2, 3\}$. Thus, $P(X) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$.

Number Theory

- Modular Arithmetic: An integer x modulo n provides the remainder of x when divided by n. For example, $3 \equiv 1 \ (mod\ 2)$, which reads "three is congruent to one modulo two". Modular arithmetic provides that if $x \equiv y \ (mod\ n)$, then $n|(x - y)$.
- Modular Inverse: $x, y \in \mathbb{Z}$ are considered modular inverses (modulo n) if $xy \equiv 1 \ (mod\ n)$.
- Quotient Remainder Theorem: $\forall n, d \in \mathbb{Z}$, $\exists q, r \in \mathbb{Z}$ (q and r are unique) such that $n = dq + r$, where $0 \leq r < d$.
- Euclidean Algorithm: Given $a, b \in \mathbb{Z}$, the Euclidean algorithm returns $gcd(a, b)$ by the following algorithm:

```
function gcd(a, b):
    while b != 0:
         temp := b
         b := a mod b
         a := temp
    return a
end function
```

- Coprime: Two integers a and b are relatively prime if and only if $gcd(a, b) = 1$.

- Chinese Remainder Theorem: Given a set of congruences with moduli that are all relatively prime to each other:
  $a \equiv x_1 \ (mod\ m_1)$
  $a \equiv x_2 \ (mod\ m_2)$
  $a\ equiv x_3 \ (mod\ m_3)$
  ...
  $a \equiv x_n \ (mod\ m_n)$
  There exists a solution modulo $M = \prod_{i=0}^{n} m_i$ of the form: $\sum_i^n x_i * \frac{M}{m_i} * (\frac{M}{m_i}^{-1} \ (mod\ m_i))$.

  As an example, let $a \equiv 3 \pmod 5$ and $a \equiv 5 \pmod 7$.

  So $a \equiv 3 * 7 * (7^{-1} \pmod 5) + 5 * 5 * (5^{-1} \pmod 7) \pmod{35}$. So $a \equiv 63 + 75 \equiv 33 \pmod{35}$.

- Pidgeonhole Principle: If there are m slots and $n > m$ objects, then there exists at least one slot with more than one object.
- Fermat's Little Theorem: Let p be prime and $a \in \mathbb{N}$. Thus, $a^{p-1} \equiv 1 \pmod p$.
- Euler-Fermat Theorem: A generalization of Fermat's Little Theorem. Given $a, n \in \mathbb{Z}$ such that $gcd(a, n) = 1$, then $a^{\phi(n)} \equiv 1 \pmod n$.

Computational Complexity

- Big-O: $f(n)$ is $O(g(n))$ if and only if $\exists C, k \in \mathbb{N}$ such that $|f(x)| \leq C * |g(x)|$, $\forall x \in \mathbb{Z} \geq k$. So 2^n is $O(3^n)$, as well as $O(n!)$ as an example. However, 2^n is not $O(n^2)$.
- Big-Omega: $f(n)$ is $\Omega(g(n))$ if and only if $\exists C, k \in \mathbb{N}$ such that $|f(x)| \geq C * |g(x)|$, $\forall x \in \mathbb{Z} \geq k$. So n^2 is $\Omega(n^2)$, as well as $\Omega(log(n))$.
- Big-Theta: $f(n)$ is $\Theta(g(n))$ if and only if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$. As an example, $2n$ is $\Theta(n)$.

Searching

- Linear Search: This algorithm traverses through a list, examining each element in sequential order until either all the elements have been evaluated or the desired key is found. Linear search runs in $\Theta(n)$ time.

```
function linearSearch(array arr, key)
  for i := 0; i < arr.length; i := i + 1
     if arr[i] == key
      return i

  return -1
end function
```

- Binary Search: This algorithm relies on the list being sorted and array-based. It partitions the list in half, starting at the middle. If the key is larger than the mid-point, it then examines the upper half of the list starting at the mid-point. Otherwise, it examines the lower half of the list in the same manner. This process is repeated until all possibilities have been exhausted or the key has been found. Binary Search runs in $O(log(n))$ time for array-based lists and $O(nlog(n))$ time for Linked Lists, as random access of elements in a Linked List is $O(n)$ rather than $\Theta(1)$ as in an array.

```
function binarySearch(array arr, key)
  low := 0
  hi := arr.length-1

  while low <= hi
    mid := (low + hi)/2
    if key < arr[mid]
      hi = mid - 1

    else if key > arr[mid]
      low = mid + 1

    else
      return mid

  return -1
end function
```

Sorting
The following swap() function will be used for the below algorithms:

```
function swap(array arr, i, j)
   temp := arr[i]
   arr[i] := arr[j]
   arr[j] := temp
end function
```

- **Selection Sort**: This algorithm works by examining a contiguous sublist of the original list. It finds the nth largest element in the list and swaps it with the nth element. It then examines n-1 terms. Selection sort runs in $\Theta(n^2)$ time for worst, bets, and average case scenarios.

```
function selectionSort(array arr)
  for i := arr.length-1; i > 0; i := i - 1
      max := i

      for j := 0; j < i; j := j+1
        if arr[j] > arr[max]
           max := j

      swap(arr, i, max)
end function
```

- **Bubblesort**: This algorithm works by swapping consecutive elements that are unordered. It examines two consecutive elements at a time. Bubblesort terminates when it completes an iteration through the list without swapping any elements. It runs in $O(n^2)$ time for average and worst cases, and $O(n)$ time if the list is already sorted.

```
function bubblesort(array arr)
     sorted := false
     while sorted == false
       sorted := true

       for i := 0; i < arr.length - 1; i := i+1
```

```
          if arr[i] > arr[i+1]
             swap(arr, i, i+1)
             sorted := true
end function
```

- **Insertion Sort**: This algorithm works by examining a larger contiguous subset of the original list on each iteration. On the first iteration, only elements at indices 0-1 are examined. If they are out of order, they are swapped. On the next iteration, elements at indices 0-2 are examined. The element at index 2 is pulled out of the list and compared against the preceding one-by-one while it is bigger than the given element. This process is repeated throughout the entire list. If the list is nearly sorted, then insertion sort runs in $O(n)$ time. Otherwise, it runs in $O(n^2)$ time.

```
function insertionSort(array arr)
  if arr.length < 2
    return

  for i := 1; i <= arr.length-1; i := i + 1
    temp := arr[i]
    j := i - 1

    while j >= 0 AND x[j] > temp
       arr[j+1] := arr[j]
       j := j - 1

   arr[j+1] := temp
end function
```

- **Radix Sort**: This algorithm works by sorting elements into buckets. It starts at the least-significant digit (the far-right digit) and places elements into buckets based on the last digit. So all elements ending with a 1 go in the same bucket, as an example. The elements are then reallocated based on the next digit to the right. This process continues until the elements have been allocated into buckets based on their most significant digits. At this point, the elements are sorted. Radix Sort runs in $O(nlog(n))$ time, where $log(n)$ is of the same base as the elements are represented. So if the elements are in bianry, the logarithm is base-2. If the elements are represented in decimal, then the logarithm is base-10.

```
function radixSort(array arr, base)
  bins := array containing base number of arrays //a 2D array
  max := -∞

  for each element in arr
     digit := element (mod base)
     bins[digit].add(element)

     if element > max
       max := element

  mostSigDigit := log_base(max)
  for i := 1; i < mostSigDigit; i := i + 1

    for each array in bins
      temp := array
      array.clear()
```

```
     for each element in temp
       digit := leastSigDigit(element, i) //get the ith least significant digit
       bins[digit].add(element)

   arr.clear()
   for each array in bins
      arr.addAll(array)

end function
```

- **Mergesort**: This algorithm partitions the list into contiguous sublists until each sublist has at most two elements. These elements are swapped if necessary. The sublists are then merged into their parent sublists such that the elements are properly ordered. Mergesort runs in $O(nlog(n))$ time, and is an ideal sorting algorithm for both arrays and linked lists.

```
function mergesort(array arr)

    if arr.length ≤ 1
      return

    mid := arr.length/2
    left := array()
    right := array()

    for i := 0; i < mid; i := i+1
       left[i] := arr[i]
       right[i] := arr[mid + i]

   mergesort(left)
   mergesort(right)
   merge(left, right, arr)
end mergesort

function merge(array left, array right, array main)
   leftIndex := 0
   rightIndex := 0
   mainIndex := 0

   while leftIndex < left.length OR rightIndex < right.length
      while left[leftIndex] ≤ right[rightIndex]
        main[mainIndex] := left[leftIndex]
        leftIndex := leftIndex + 1
        mainIndex := mainIndex + 1

      while right[rightIndex] ≤ left[leftIndex]
        main[mainIndex] := right[rightIndex]
        rightIndex := rightIndex + 1
        mainIndex := mainIndex + 1
end function
```

- **Quicksort**: This algorithm works to sort arrays by ordering elements around a partition point. Elements larger than the parttion go to the right of it, and elements smaller than the partition go to the left of it. Elements equal to the partition value can be placed on either side, as long as this is handled consistently. The subsets on either side of the partition are then recursively

evaluated until contiguous subsets of no more than two elements are evaluated.

```
function quicksort(array arr, start, end)
     if start < end
        partitionIndex := partition(array, start, end)
        quicksort(arr, start, partitionIndex-1)
        quicksort(arr, partitionIndex+1, end)
end function

function partition(array arr, start, end)
   pivot := (start + end)/2
   swap(arr, end, pivot)

   partitionPoint := start

   for i := start; i < end; i := i + 1
      if arr[i] ≤ arr[end]
        swap(arr, i, partitionPoint)
        partitionPoint := partitionPoint + 1

   swap(arr, partitionPoint, end)
   return partitionPoint
end function
```

- **Heapsort**: This algorithm works by constructing a max-heap, or equivocally a min-heap. The largest (or smallest) element is then removed from the heap and inserted into the array. The heap is then reassembled based on the removal. This process is repeated until the heap is empty. Heapsort runs in $O(nlog(n))$ time.

```
function heapsort(array arr)
    heapify(arr, arr.length)
    end := arr.length - 1

    while end > 0
      swap(arr, 0, end)
      end := end - 1
      restoreHeap(arr, 0, end)
end function

function heapify(array arr, length)
  for i := (length - 2)/2; i ≥ 0; i := i - 1
    restoreHeap(arr, i, length-1)
end function

function restoreHeap(array arr, begin, end)
   rootIndex := begin

   while 2 * root + 1 ≤ end
     child := 2 * root + 1
     if child + 1 ≤ end AND arr[child] < arr[child+1]
       child := child + 1
    if arr[root] < arr[child]
       swap(arr, root, child)
       root := child
```

```
    else
       return
end function
```

Series

Binomial Theorem: Let $f(x) = (a+x)^n = \sum_{i=0}^{n} \binom{n}{i} x^i a^{n-i}$

Convergent Geometric Series: $\sum_{i=0}^{\infty} = ar^i = \frac{a}{1-r}$ (whenever $|r| < 1$).

Sum of the first n terms in a Geometric Series: $\sum_{i}^{n} ar^i = a * \frac{1-r^{n+1}}{1-r}$

Harmonic Series: $\sum_{i}^{n} \frac{1}{i}$ (diverges).

P-Series: $\sum_{i}^{n} \frac{1}{i^p}$ converges only when $p > 1$.

Telescoping Series: $\sum_{i=0}^{n} a_i - \sum_{j>i}^{n} a_j = \sum_{i=0}^{j-1} a_i$