## 1.1 - Algorithms

Notation:

* Define function name
* Define function input parameters
* Define output
* Define list of instructions forming algorithm
  + Detail based on problem modelled



## 1.1 - Random-access machine (RAM) model

Characterizes algorithm’s complexity

Details:

* Unbounded number of memory cells
* Single CPU
* Computes number of unit times required for running algorithm
  + Each instruction costs 1
  + Count total number of operations
  + Not worst case, exact depending input



Above is therefore T(n) = 4n+3

Recursively:



## – Asymptotic Notation

Why?

* RAM forces us to do laborious analysis of algorithm’s steps
* Most instructions do not depend on input and are less insightful to analyze

Asymptotic notation allows us to focus on operations that depend on size only

Big-Oh notation

* f(n) and g(n) continuous functions mapping nonnegative integers to real numbers
* f is big-oh(g) / f is O(g) = “f grows no faster than g past a certain point”
  + Upper bound (worst case) of algorithm f(n)’s time required is O(f(n))
* f is big-theta(g) / f is Θ(g) = “f grows at exactly same rate of g past a certain point”
  + Best case of all the worst-case times that the algorithm can take
* f is big-omega(g) / f is Ω(g) = “f grows at least as fast as g past a certain point”
  + Lower bound (best case) of algorithm f(n)’s time required
* f(n) is O(g(n) if there is a real constant c>0 and an intenger constant n0 >= 1 such that f(n) <= cg(n) for every integer > n0
  + “As n goes to infinity, f(n) > g(n)”
* Big-Oh rules given d(n), e(n), f(n), g(n) mapping nonnegative integers to nonnegative reals:
  + Case 1: If d(n) is O(f(n)), then ad(n) is O(f(n)) for any a > 0
    - If n2 is O(n2), then 2n2 is O(n2)
  + Case 2: If d(n) is O(f(n)) and e(n) is O(g(n)), then d(n) + e(n) is O(f(n) + g(n))
    - If 2n3 is O(n3) and 4nlogn is O(n2), then 2n3+4nlogn is O(n3+n2)
  + Case 3: If d(n) is O(f(n)) and e(n) is O(g(n)), then d(n)e(n) is O(f(n)g(n))
    - If 2n3 is O(n3) and 4nlogn is O(n2), then (2n3)(4nlogn) is O(n5)
  + Case 4: If d(n) is O(f(n)) and f(n) is O(g(n)), then d(n) is O(g(n))
    - If 2logn is O(logn) and logn is O(n) then 2logn is O(n)
  + Case 5: If f(n) is a polynomial of degree d, then f(n) is O(nd)
    - If f(n) == 2n3+4nlogn, then 2n3+4nlogn is O(n3)
  + Case 6: nx is O(an) for any fixed x > 0 and a > 1
    - n100 is O(2n)
  + Case 7: log(nx) is O(logn) for any fixed x > 0
    - log(n100) is O(logn)
  + Case 8: logxn is O(ny) for any fixed constants x>0 and y>0
    - log100x is O(n)
* F(n)s ordered by O(f(n)) ascending
  + Logn
  + Log2n
  + Sqrt(n)
  + N
  + Nlogn
  + N2
  + N3
  + 2n

# 1.4 – Amortized Analysis

Estimates time required to perform sequence

We have opportunity to show that the average cost of an operation is small, even if the cost of such operation looks big in asymptotic notation

Two techniques

* Accounting: Faster to use and intuitive, better for simple operations
* Potential: More formal and structured, better suited for involved algorithms

## Accounting

* Identify operations algorithm will perform, which are cheap, which are expensive
  + Cheap require less than average
  + Expensive require more than average
* For each operation in sequence:
  + If cheap, increment credit by cost of operation
  + If expensive, subtract credit from operation to determine actual cost
    - Actual cost is difference between cost of operation and credit
* At end, divide total cost by number of operations to determine average cost per operation
* Eg: If in the worst case scenario, operation x (k real cost) necessitates prior operation y (1 real cost) k times, then operation y is 2 amortized cost, and operation x is 0 amortized cost

## Potential

Choose a function **Φ(S) that maps a functions current state into non-negative values**

* **Can be thought of as computing quantity of potential energy that the state has to provide**
* **Set to 0 before data structure initialized**
* **Φ(a0) = 0 where a0 is starting state**
* **Φ(at) >= 0 for all states of data structure occurring at time of course of computation**
* **Despite calculating amount of time that can be saved up to cover expensive operations, it merely depends on the data structure’s current state regardless of the history that led to it**

**Amortized cost of operation = real cost + Φ(Data structure after given operation) − Φ(Data structure before given operation)**

* Amortized time is calculated as actual time plus prospective change
  + Amortized time should ideally be low when defined

# Union-Find Data Structures

Union and find are two operations involved in number of applications

* Social network analysis, containment relationship, percolation theory

Always has three operations:

* MakeSet(e) – create singleton with one element e
* Union(A,B) – Add set A to set B (or opposite depending on implementation)
* Find(e) – Return name of set containing e

## List-based implementation



## Tree-based implementation

Merging two trees is very fast, O(c)

Finding root of set of node more involved:

* With no optimization, O(n)
* With union-by-size we can decrease complexity to O(logn)
* With path-compression we can decrease to inverse of ackerman function
  + Any sequence of m union and find on initial set of n is O(m \* α(m,n))
    - Amortized cost of each function O(α(m,n))
    - α(m,n)) =

Base implementation:



Union-By-Rank implementation:



Path compression implementation



# Trees

Bin