ECE 592
Topics in Data Science

Dror Baron
Associate Professor
Dept. of Electrical and Computer Engr.
North Carolina State University, NC, USA

Computational Complexity

[Cormen et al., Chapter 1, 2.1]

Keywords: algorithms, complexity, growth of functions

Algorithms and Running Time

[Cormen et al., Chapter 1]

Keywords: algorithms, running time

A weird example

- Let's fill an array with values [1 2 … 9999]

```
x=[];
for n=1:9999;
  x=[x n];
end
```

- This code is s l o w . . .
- Why?

Let’s back up a bit…

- Wikipedia:

In mathematics and computer science, an **algorithm** is a self-contained step-by-step set of operations to be performed. Algorithms perform calculation, data processing, and/or automated reasoning tasks.

Algorithms

- Algorithms convert inputs into outputs
- Could have different algorithms for same conversion (e.g., discrete Fourier transform vs. fast Fourier)
- Could have different implementations of same algo

Analysis of algorithms

- Want to predict resources used by algorithm

- What resources?
  - Running time
  - Memory consumption
  - Communication requirements
  - Number of logic gates
  - Power consumption

What sort of analysis?

- What sort of computer?
  - Different machines vary drastically, right?
  - *Random access machine model* – instructions executed sequentially
- Want our analysis to express main characteristics of resource consumption
  - And ignore minor stuff
- Primary focus on running time

Runtime may depend on input

- Q: How to analyze algorithms whose running time depends on input?
- A: worst case, average case, & best case

- Worst case often of greatest interest
  - Guarantee on runtime
  - Worst case might happen often
  - Worst case and average case might be similar

How to measure runtime?

- Want running time as function of input size

- Input size
  - Could be # items in input
  - Could be # bits to represent input
  - Could be multiple parameters (matrix: #rows, #columns)

- Measuring running time
  - Number of steps executed
  - Random access machine → const time per line
  - Calling a routine – one line; running it could be more

Order of Growth

[Cormen et al., Chapter 1.2]

Keywords: growth of functions

Two sorting algorithms

- Insert sort
  - Maintain (sorted) list of numbers processed so far
  - Next item gets inserted into list

- Merge sort
  - Divide problem into two parts (roughly equal size)
  - Conquer each problem (run merge sort recursively)
  - Merge solutions

Example

- Let’s run insert sort and merge sort
- Input x=(1, 4, 2, −3, 7, 2, 10, 5)

Their running times

- Running time T(n) when sorting n numbers
  - Insert sort: $T_i(n)=n^2$
  - Merge sort: $T_m(n)=n\times\log_2(n)$

- Let's give insert sort an edge
  - Merge implemented by bad programmer → $100n\times\log_2(n)$
  - Insert runs on cluster (10^{12} floating point operations/sec [flops])
  - Merge runs on regular machine (10^9)

Running times continued

- $n=10^3$
  - Merge sort: $100n \times \log_2(n)/10^9$ flops = 1 ms
  - Insert sort: $n^2/10^{12}$ = 1 us
- $n=10^6$
  - Merge sort: $100n \times \log_2(n)/10^9$ flops = 2 s
  - Insert sort: $n^2/10^{12}$ = 1 s
- $n=10^9$
  - Merge sort: $100n \times \log_2(n)/10^9$ flops = 3000 s (50 minutes)
  - Insert sort: $n^2/10^{12}$ = 11 days
- *Asymptotic growth matters*

Order of growth

- Consider $T(n)=an^2+bn+c$
  - a, b, c positive constants

- Asymptotically, an^2 matters
  - bn+c doesn't

- Need to characterize asymptotic growth → complexity

Formal Notions of Complexity

[Cormen et al., Chapter 2.1]

Keywords: computational complexity

Different types of computational complexity

- Computational complexity = formal classification of functions based on rate of asymptotic growth

- Different types of growth (details coming up)
  - $f(n)=\Theta(g(n))$ tight asymptotic bound
  - $f(n)=O(g(n))$ upper bound for f(n)
  - $f(n)=\Omega(g(n))$ lower bound for f(n)
  - $f(n)=o(g(n))$ ratio f(n)/g(n) vanishes

Asymptotically tight growth

- Size of input n (natural number)
- f(n), g(n) positive

- $\Theta(g(n)) = \{f(n): \exists c_1, c_2, N_0 > 0 \text{ s.t. } 0 < c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n > N_0\}$

- $f(n) = \Theta(g(n))$ means f(n) in class of functions that grow as fast as g(n)

- Main idea – can ignore lower order terms

Example

- Let's show formally that $n^2-3n=\Theta(n^2)$
- Need to find c_1, c_2, N_0

More definitions

- $\Theta(g(n)) = \{f(n): \exists c_1, c_2, N_0>0 \text{ s.t. } 0 < c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n>N_0\}$

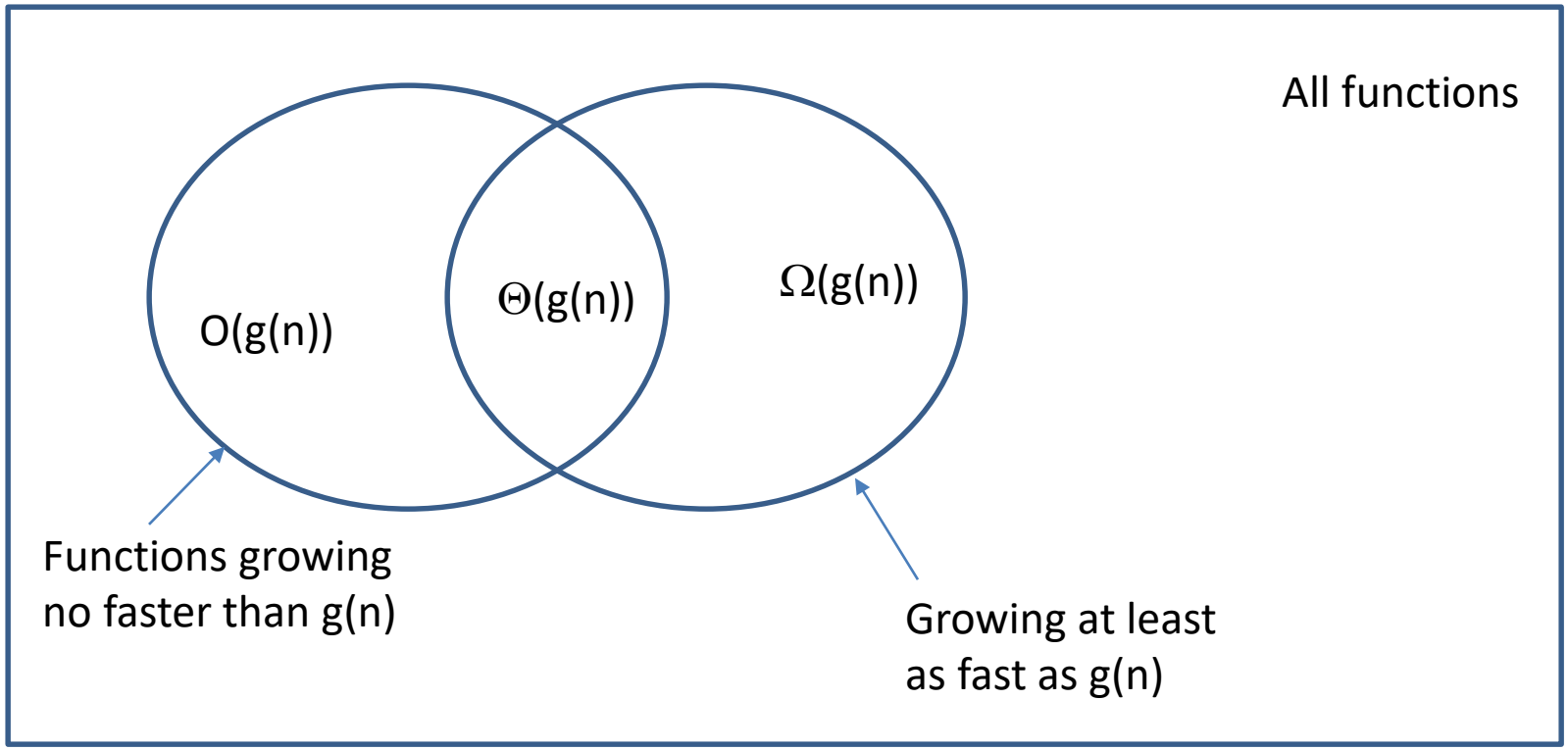
- $O(g(n)) = \{f(n): \exists c, N_0>0 \text{ s.t. } 0 < f(n) \leq cg(n), \forall n>N_0\}$
  - Pronounced “Big O”
  - Asymptotic upper bound

- $\Omega(g(n)) = \{f(n): \exists c, N_0>0 \text{ s.t. } 0 < cg(n) \leq f(n), \forall n>N_0\}$
  - Asymptotic lower bound

- $f(n)=o(g(n))$ means $\lim_{n\to\infty} f(n)/g(n)=0$
  - Pronounced “little o”

Intuition

- $f(n)=\Theta(g(n))$ if and only if $f(n)=O(g(n))$ and $f(n)=\Omega(g(n))$



Selecting Algorithms via Complexity

[Cormen et al., Chapter 2.1]

Keywords: computational complexity

Some low-complexity examples

- $\Theta(1)$ – run few simple lines of code
- $\Theta(\log(n))$ – searching for element in balanced tree data structure (will learn)
- $\Theta(n^{0.5})$ – determine whether a number is prime

Medium-complexity examples

- $\Theta(n)$ – find min/max among n numbers
- $\Theta(n \times \log(n))$
  - Sort n numbers
  - Fast Fourier transform (FFT)
- $\Theta(n^2)$
  - Matrix vector product ($n \times n$ matrix)
  - Direct computation of discrete Fourier transform (DFT)
- $\Theta(n^3)$ – matrix inversion

High-complexity examples

- $\Theta(2^n f(n))$ – optimally decode n bits (communication)
  - f(n) – running time to evaluate each n-tuple

- $\Theta(n! f(n))$ – process all permutations of n objects
  - f(n) – evaluate each permutation

How to select between algorithms?

- If two algorithms have “quite different” complexities, choose lower

- Examples:
  - Use FFT to compute Fourier transform
  - Prefer merge sort over insertion sort

- What if complexities are similar?

Counter example [B & Bresler, 2005]

- Suffix sorting – used in some data compression algorithms
- Various implementations

- Previous approaches:
  - Suffix trees – linear worst case, $\Theta(n)$
  - Fastest methods in practice – linear average case, quadratic worst, $O(n^2)$
  - For one "bad" text file (< 1 MB), "fastest" method required almost an hour; suffix trees ran in 4-5 seconds

Counter example [B & Bresler, 2005]

- New algorithm proposed

D. Baron and Y. Bresler, "Anti-Sequential Suffix Sorting for BWT-Based Data Compression," IEEE Trans. Computers, vol. 54, no. 4, pp. 385-397, Apr. 2005

- Computational complexity $\Theta(n \times \log^{0.5}(n))$
  - Faster than suffix trees (sqrt(log(n)) is small)
  - Reasonable worst case

- *Constants matter unless computational complexity quite different*

Algorithm Design

[Cormen et al., Chapter 1.3]

Keywords: divide and conquer, recursion

Divide and conquer approach

- Many computational problems can be approached as follows

1. *Divide* problem into sub-problems
2. *Conquer* each sub-problem recursively
3. *Combine* solutions

- Note: if problem is small enough, solve directly; apply recursion to sub-problems only if big enough

- Examples: merge sort, FFT, …

Running time of divide and conquer

- Direct solution of small problems is $\Theta(1)$
- Dividing size-n problem into a sub-problems of size n/b: D(n)
- Combining into size-n solution: C(n)

- Recursive formula:

$$T(n) = \begin{cases} D(n) + aT\left(\frac{n}{b}\right) + C(n), & n \geq N_0 \\ \Theta(1), n < N_0 \end{cases}$$

Example (Question 4, practice midterm 2016)

- Suppose that merge sort runs in $64n \times \log_2(n)$ steps while insertion sort takes $8n^2$
  - For which value of n does merge sort start beating insertion sort?
  - How to modify merge sort to obtain faster performance on small inputs? Discuss the modification and new runtime.

Typical Computational Architectures

Keywords: cache, GPU, memory hierarchy, multi processor

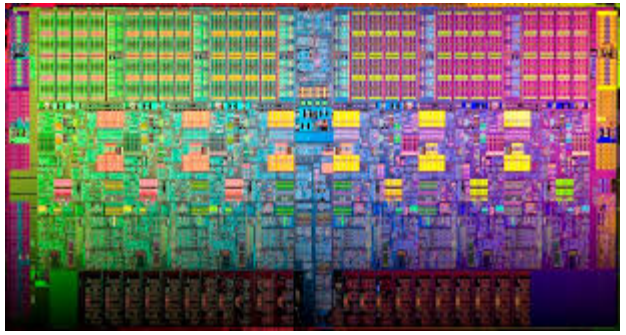
Why consider computational architecture?

- Random access machine model somewhat simple
- Some modern architectures offer significant speedups (2+ orders of magnitude) via parallelization
- Advantageous to be aware of opportunities

Types of processors

- Low-end embedded processor
  - Clockspeed several MHz
  - Memory < 1 MB
  - Limited instruction set → math operators require many clock cycles

- Typical central processing unit
  - Examples: Intel/AMD laptop/desktop, Intel Xeon server, smartphone
  - Clockspeed 2-5 GHz
  - Memory in GB (could be hundreds)
  - Fast math operations
  - Many billions of transistors



Intel Xeon

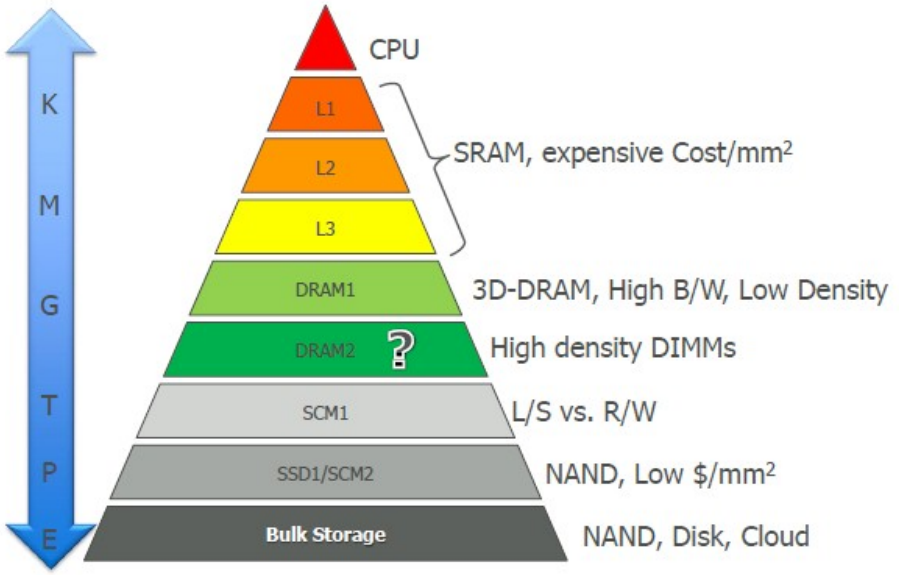
Multi-processors

- Some systems/chips support multiple processors
- Widespread – Intel/AMD chips with multiple cores
- General purpose graphics processing unit (GP GPU)
  - Initially designed for graphics processing
  - Highly parallelizable
  - Currently support up to *thousands* of cores
  - Much faster but constrained (not fully parallel)
- Clusters (cloud computing)

Nvidia Tesla K80

Memory hierarchy

- Main idea – fast memory is expensive
  - Partition memory into several hierarchies
  - Top of pyramid – small amount of fast memory
  - Bottom – large amount of cheap slow memory
  - Search for data in top of pyramid, else spill into lower levels



Types of memory in hierarchy

- Registers – several dozen; on CPU; same-clock access
- Cache – several MB; 1-dozens clocks
- Main memory (RAM) – several GB; ~100 clocks
- Permanent memory (disk, cloud?) – TB; slow

Memory in GPUs

- GPU have significant (GBs) on-chip memory
- Each core has small fast local memory
- GPU chip has significant slower memory
  - Challenge: Could be very slow for each core to access memory
  - Solution: hardware support for adjacent memory access with high bandwidth (hundreds of GB/second) interconnect
- *Bottom line – solid GPU programming is tough*

Parallel Processing

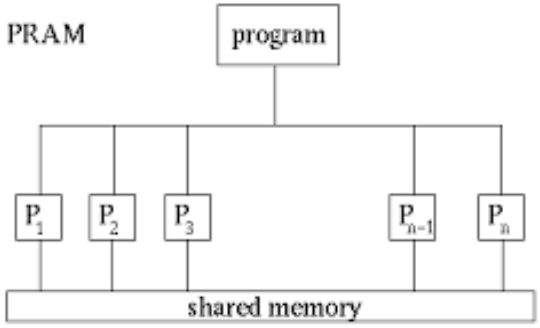
[Cormen et al., Chapter 30]

Keywords: parallel computers, parallel random access machine (PRAM)

Parallel random access machines (PRAM)

- Recall random access machine (RAM) model
  - Serial (not parallel)

- Want model for parallel RAM (PRAM) machine
  - Parallel architectures are quite intricate → want to capture main stuff
  - Assume that time equates to # parallel memory accesses
  - *Imprecise assumption – access time grows with # processors p*



Types of PRAM memory access

- Concurrent read – PRAM algo reads concurrently (simultaneously) from same location
- Exclusive read – never read same memory location concurrently

- Same for concurrent/exclusive write

- Types of PRAM machines:
  - EREW – exclusive read exclusive write
  - CREW – concurrent read exclusive write
  - ERCW – exclusive read concurrent write
  - CRCW – concurrent read concurrent write

Discussion

- CRCW PRAM supports EREW algos
  - Not vice versa

- EREW – simple hardware → fast
- CRCW – complicated hardware → slow

- Synchronization between cores can be messy

- CRCW algos sometimes have lower computational complexity than EREW (but worse constants)

Data Structures

[Cormen et al., Chapter 11]

Keywords: arrays, data structures, linked lists, queues, sets, stacks

Why do we need data structures?

- Want to organize data efficiently
  - Data is *set* of objects/elements
  - Low memory footprint
  - Want fast access/searches
  - Want fast *updates*

- Want to support *dynamic sets*
  - Changes over time
  - Key operations: insert, delete, check membership
  - If we want more operators, need more refined data structure

What does data structure need to support?

- Data arranged in objects that contain fields
  - Key – field that identifies objects
  - Other fields contain attributes about object

- Common operators
  - Search(S,k) – searches for object with key k in set S
  - Insert(S,x) – x is object
  - Delete(S,x) – needs pointer to x (not its key)
  - Minimum(S) – returns smallest key
  - Maximum(S) – largest key
- For ordered sets:
  - Successor(S,x) – next object in structure; NIL if already last/largest
  - Predecessor(S,x) – previous object; NIL if first/smallest

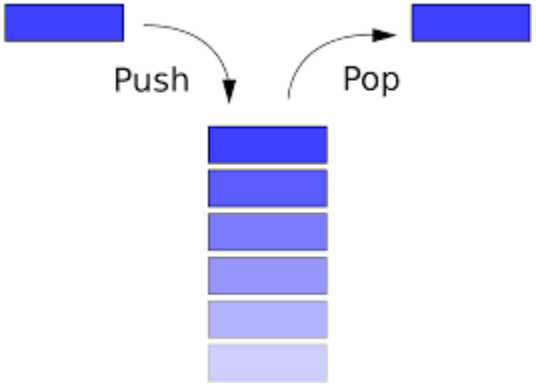
Stacks and Queues

[Cormen et al., Chapter 11.1]

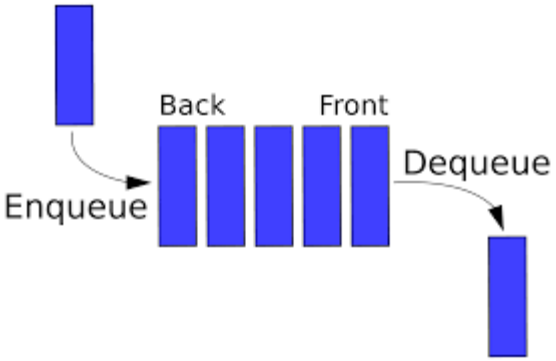
Keywords: queues, stacks

Stacks vs. queues

- Stack
  - Always remove last element that was inserted
  - Last in first out (LIFO)
  - Push (insert) new object onto stack
  - Pop (delete) old one
  - Application – operating system stores list of routines we call in stack; when exiting routine, remove info about last one (current routine)



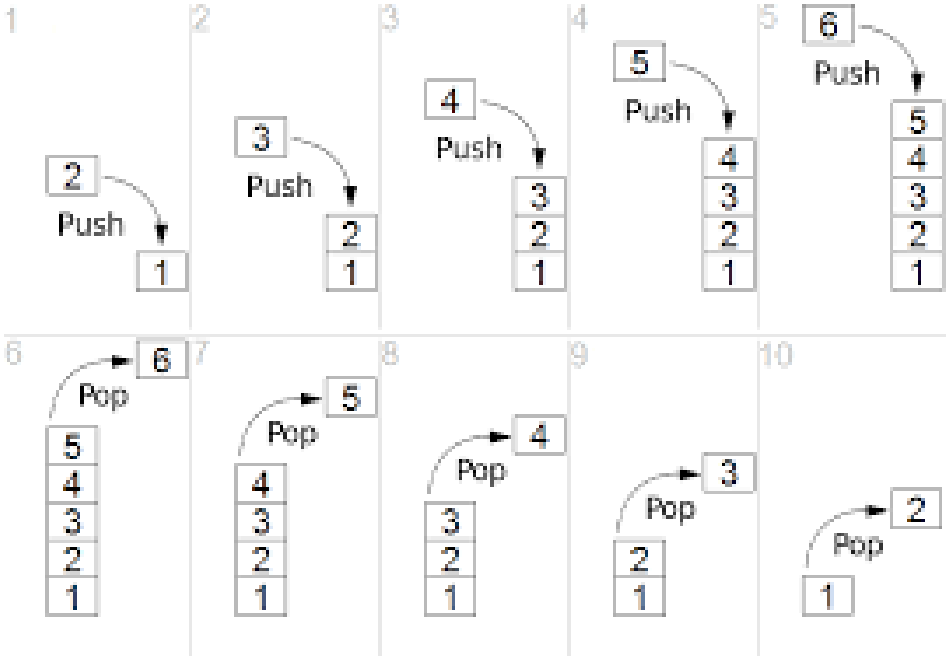
- Queues
  - Always remove first element that was inserted
  - First in first out (FIFO)
  - Enque (insert) and dequeuer (delete)
  - Application – customers waiting for their requests to be processed



Implementing stacks

- Implement as array S[1,…,n]
  - Advantage: simple
  - Disadvantage: could have overflow
  - Must store Top(S)

- Operators
  - Stack_empty(S)
  - Stack_full(S)
  - Push(S,x)
  - Pop(S)



Implementing queues

- Implement as array Q[1,…,n]
  - Store Head(Q) and Tail(Q) (back/front of queue)
  - Elements in queue: Head(Q), Head(Q)+1, …, Tail(Q)-1
  - Indexing is modulo-n
  - Head(Q)=Tail(Q) → queue empty
  - Head(Q)=Tail(Q)+1 → queue full

- Operators
  - Enqueue – store data, increment Tail(Q)
  - Dequeue – retrieve data, increment Head(Q)

Linked Lists

[Cormen et al., Chapter 11.2]

Keywords: linked lists

What does list do?

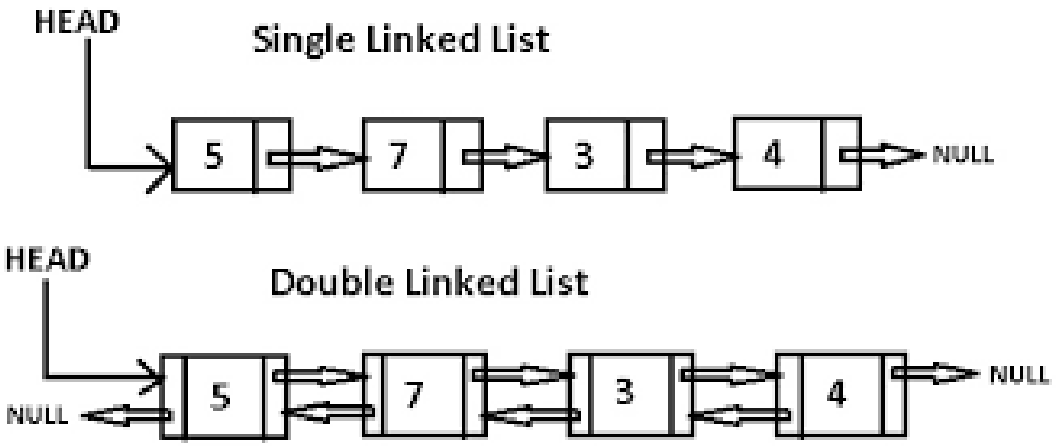
- Main objective – arrange objects in linear order

- Arrays
  - Objects ordered using index (integer)
  - Difficult to add object “in the middle” (what does index 3.6 mean?)

- Lists
  - Objects arranged with pointers
  - Easy to insert/delete objects by updating pointers

Types of lists

- Doubly linked list
  - Each object contains key, pointers to next/prev



- Single linked – only next pointer (no prev)

- Sorted vs. unsorted (easier to search through sorted)

Operators on linked lists

- List_search(L,k)
  - Search for key k in list
  - Complexity O(n) not Θ(n)
- List_insert(L,x)
  - Adds new object to head of list; Θ(1)
- List_delete(L,x)
  - Must splice off data structure

Graphs and Trees

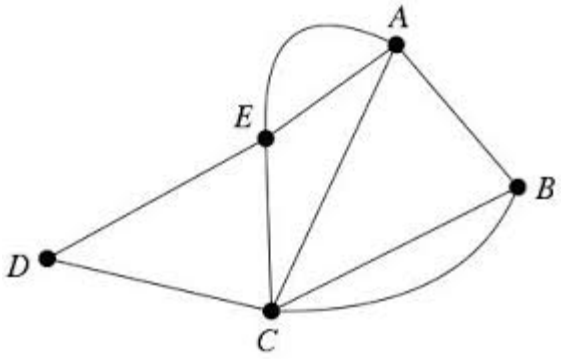
[Cormen et al., Chapter 5.4-5.5]

Keywords: graphs, trees

What's a graph?

- Structure relating different objects

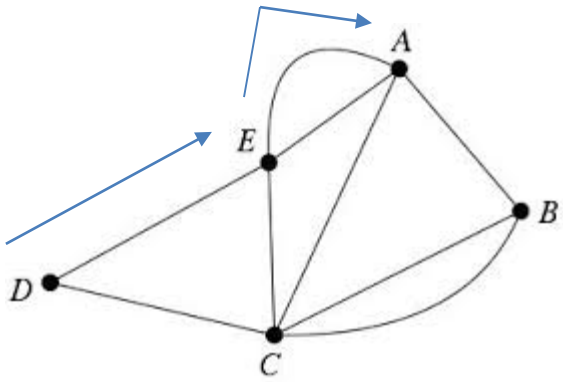
- G(V,E)
  - Graph G
  - Vertices V (also called nodes)
  - Edges E (between two vertices)



- Can be
  - Directed graph - edges are arrows
  - Undirected

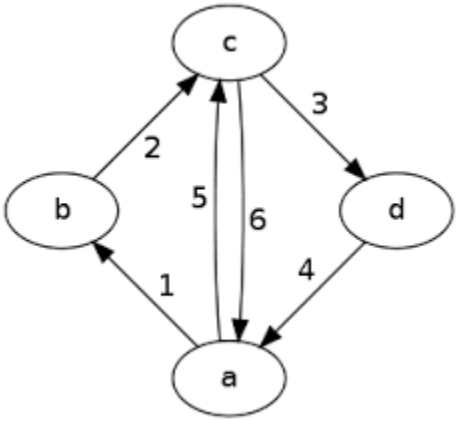
Concepts

- Consider edge $(u,v) \in E$ where $u,v \in V$
  - We say v adjacent to u
- Degree(v) = # edges connecting to vertex v
- Length-k path p from u to u'
  - Edges $(v_0,v_1), (v_1,v_2),\ldots, (v_{k-1},v_k)$
  - $v_0=u$, $v_k=u'$, $(v_{i-1},v_i) \in E$, $i \in \{1,\ldots,k\}$
  - u' reachable from u using path p
- Example: length-2 path p=(D,E),(E,A)



More about paths

- Simple path – all vertices on path are distinct
  - Not distinct → can shorten path
- Cycle – path starts/ends same vertex
  - Examples: p_1=(a,b),(b,c),(c,a), p_2=(a,c),(c,a)



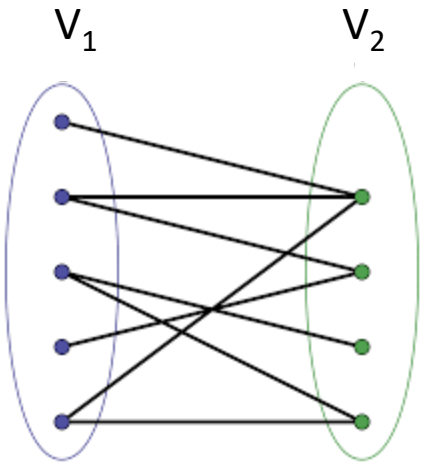
- Acyclic graph – graph without cycles

Connectivity in graphs

- Undirected graph
  - Connected component – all nodes reachable from one another
  - Connected components partition V into equivalent classes
  - Connected graph – has one (large) connected component
- Directed graph
  - Strongly connected – all nodes reachable (via directed paths) from one another
- Complete graph – all vertex pairs are adjacent

Bipartite graph

- V can be partitioned into V_1, V_2
- $(u,v) \in E$ implies
  - Either $u \in V_1$ & $v \in V_2$
  - Or $v \in V_1$ & $u \in V_2$



- Application: linear regression $Y=X\beta+\varepsilon$
  - V_1 corresponds to Y
  - V_2 corresponds to β
  - Matrix X corresponds to edges E
  - Estimate β by passing messages between V_1 and V_2
  - Details: B, Sarvotham, & Baraniuk, "Bayesian Compressive Sensing via Belief Propagation," IEEE Trans. Signal Proc., Jan. 2010

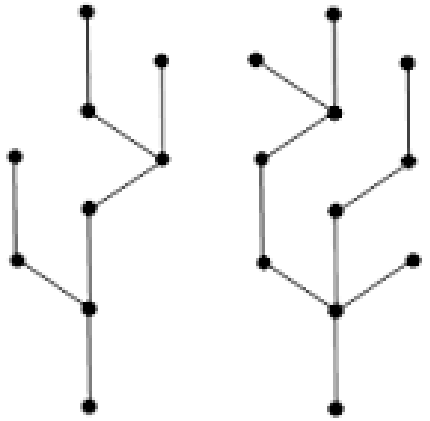
Trees

[Cormen et al., Chapter 5.5]

Keywords: acyclic graphs, forests, free trees, rooted trees

Forests and trees in undirected graphs

- Forest = acyclic *undirected* graph
- Different components connected without cycles



- Tree = connected forest
  - Or forest = union of trees

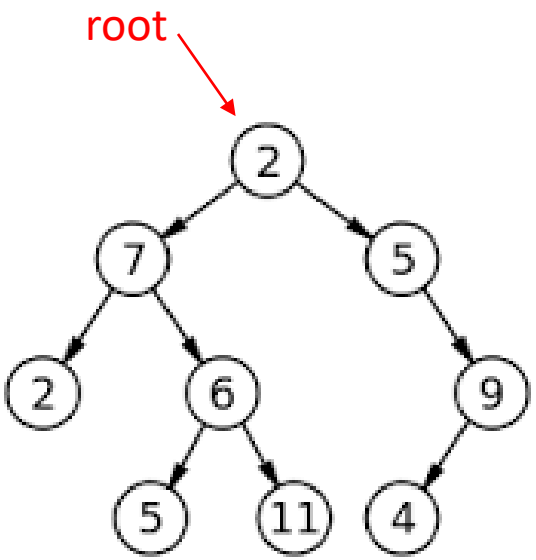
- Are acyclic graphs good?
  - Redundant edges could be costly → good
  - No connectivity if edge "breaks" → not robust → bad

Properties of trees

- Theorem: undirected G(V,E), following are equivalent
  - G is tree
  - Any $v_1, v_2 \in V$ connected by unique simple (no cycles) path
  - G connected & removing any edge makes it disconnected
  - G connected & $|E|=|V|-1$
  - G acyclic & $|E|=|V|-1$
  - G acyclic & adding any edge creates cycle

Free trees vs. rooted trees

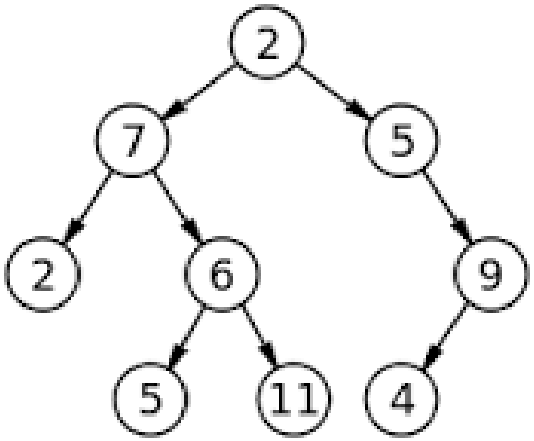
- Directed graph
  - Rooted tree - one of nodes is root
  - Paths lead *from* root *to* other nodes
  - Example: node 2 is root



- Earlier we considered undirected graph
  - Free trees
  - No concept of from/to

More about rooted trees

- Path from root r to node x is unique
  - Node y on path is ancestor of x
  - x descendant of y
  - Example: node 11 is descendant of node 7



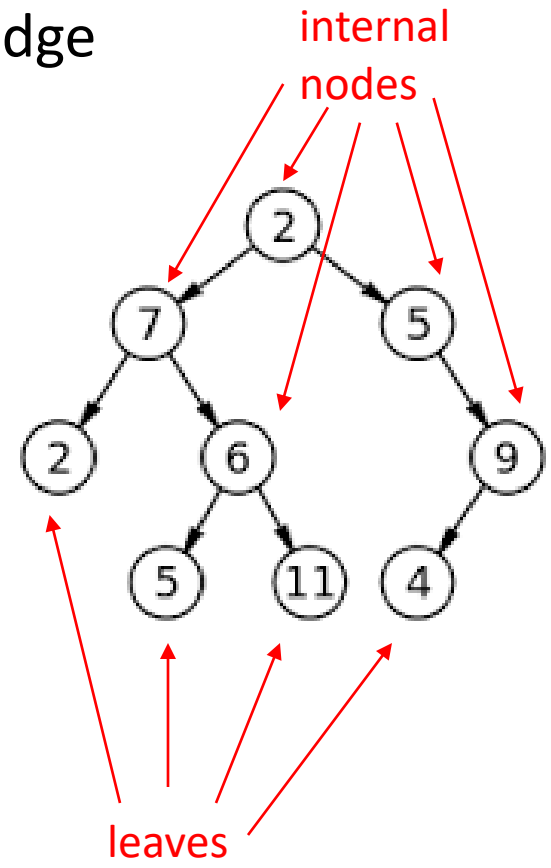
- Subtree at x = tree induced by descendants of x
  - Example: subtree of 7 = {7,2,6,5,11}

- Depth(x) = length of path from r to x
- Height(T) = maximal depth among all nodes

Children and parents

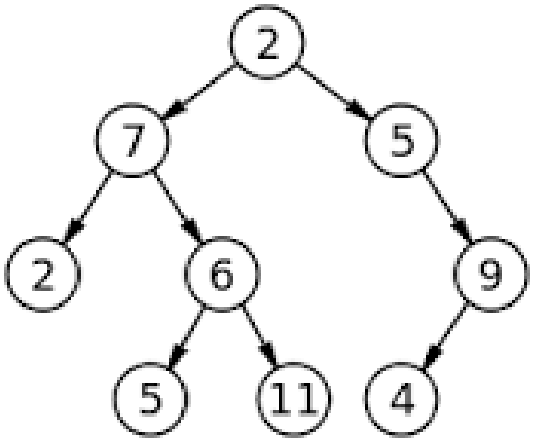
- Consider x descendant of y & connected by edge
  - x child of y
  - y parent of x

- Properties
  - All nodes except r have single parent
  - Leaf = node without children
  - Internal node = not leaf



Implementing trees

- Details vary based on type of tree
  - Fixed # children per node?
  - Ordered or not?



- Typical approach
  - Each node contains pointers to child/children, parent, sibling node(s), parent node, various fields
  - Pointer to root

New Example

- Consider an undirected acyclic graph G(V,E) with |V|=6 vertices and |E|=4 edges
- Sketch a possible such graph; is it a tree?

Putting it Together

Keywords: coding, profiling

Our assignment

- Will develop a merge sort routine

- Main structure:

```
mergesort(input x, output y)
if x is short
    y=x
else
    y1=mergesort(first half) % recursive call
    y2=mergesort(second half)
    y=merge(y1,y2) % merge both halves
end
```

How to implement merge?

- Input vectors x1, x2
- Loop over:
  - Compare first numbers in both vectors
  - Move smaller one into output array; increment pointer(s) accordingly
- Are length(x1) and length(x2) same?

Profiling

- Wikipedia:

In software engineering, profiling ("program profiling", "software profiling") is a form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. Most commonly, profiling information serves to aid program optimization.

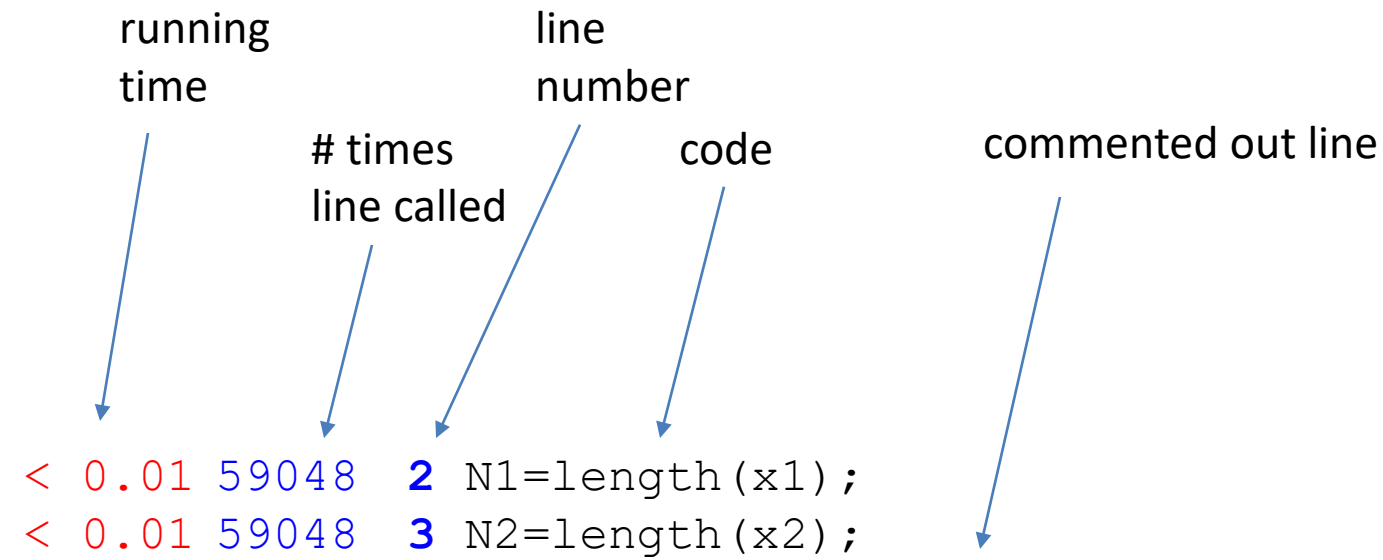
Profiling continued

- Profiling measures running time consumed on each line/function
- Number of times each line/function ran

- Matlab mini-example:

```
x=randn(23,1);
profile on
y=mergesort(x);
profreport % generates detailed report
```

Typical profiling report




```
< 0.01 59048   2 N1=length(x1);
< 0.01 59048   3 N2=length(x2);
               4 %y=zeros(N1+N2,1); % initialize
< 0.01 59048   5 index1=1; index2=1; % where we're pointing into
< 0.01 59048   6 for n=1:N1+N2
0.05   862117  7 if x1(index1)<x2(index2) % first element is
0.18   425863  8 y(n)=x1(index1);
0.01   425863  9 index1=index1+1;
0.02   425863 10 if index1>N1 % ended processing x1
```

Profiling methodology

- Look through all lines with substantial running time
- Make sure you know why it took plenty of time
- Re-design as needed