Algorithms for Big Data Worked Problems

Topics

We will be using the lecture slides for the Kleinberg and Eva Tardos book provided by Addison Wesley at http://www.cs.princeton.edu/~wayne/kleinberg-tardos/

- What is an algorithm?
- Algorithm Analysis (big-Oh notation)
- Graphs (graph search)
- Greedy Algorithms (shortest paths and MSTs)
- Divide and Conquer (sorting and selection)
- Dynamic Programming (basic techniques. sequence alignment, Bellman-Ford)
- Network Flow (maximum flow theory, maximum flow applications, assignment problem)
- Intractability (polynomial-time reductions, P, NP, and NP-complete)
- Approximation Algorithms (approximation algorithms)
- Local Search (Metropolis, Hopfield nets)
- Randomized Algorithms (randomized algorithms)

What is an algorithm?

In mathematics and computer science, an algorithm is an unambiguous specification of how to solve a class of problems. Algorithms can perform calculation, data processing and automated reasoning tasks.

An algorithm is an effective method that can be expressed within a finite amount of space and time and in a well-defined formal language for calculating a function. Starting from an initial state and initial input (perhaps empty), the instructions describe a computation that, when executed, proceeds through a finite5 number of well-defined successive states, eventually producing "output" and terminating at a final ending state. The transition from one state to the next is not necessarily deterministic; some algorithms, known as randomized algorithms, incorporate random input.

The concept of algorithm has existed for centuries; however, a partial formalization of what would become the modern algorithm began with attempts to solve the Entscheidungsproblem (the "decision problem") posed by David Hilbert in 1928. Subsequent formalizations were framed as attempts to define "effective calculability" or "effective method".

Algorithm Analysis (big-Oh notation)

Big O notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity. It is a member of a family of notations invented by Paul Bachmann,1 Edmund Landau,2 and others, collectively called Bachmann–Landau notation or asymptotic notation.

In computer science, big O notation is used to classify algorithms according to how their running time or space requirements grow as the input size grows.3 In analytic number theory, big O notation is often used to express a bound on the difference between an arithmetical function and a better understood approximation; a famous example of such a difference is the remainder term in the prime number theorem.

Big O notation characterizes functions according to their growth rates: different functions with the same growth rate may be represented using the same O notation.

The letter O is used because the growth rate of a function is also referred to as order of the function. A description of a function in terms of big O notation usually only provides an upper bound on the growth rate of the function. Associated with big O notation are several related notations, using the symbols , to describe other kinds of bounds on asymptotic growth rates.

Big O notation is also used in many other fields to provide similar estimates.

Formal definition

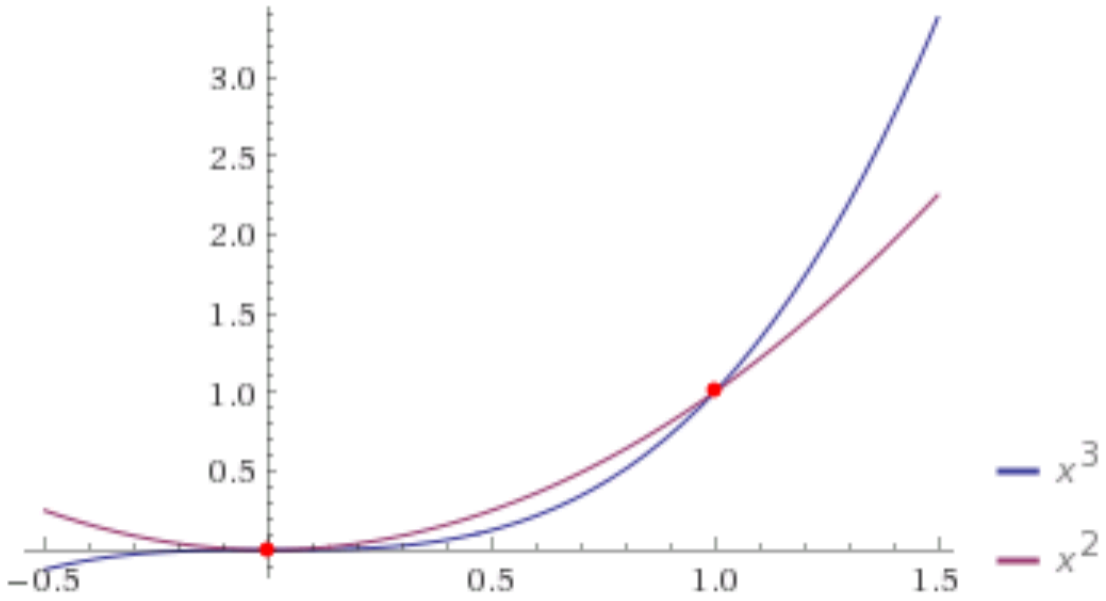
Let *f* and *g* be two functions defined on some subset of the [real numbers]. One writes

$$f(x)=O(g(x)) \text{ as } x\to\infty f(x)=O(g(x)) \text{ as } x\to\infty$$

if and only if there is a positive constant *M* such that for all sufficiently large values of *x*, the absolute value of *f*(*x*) is at most *M* multiplied by the absolute value of *g*(*x*). That is, *f*(*x*) = *O*(*g*(*x*)) if and only if there exists a positive real number *M* and a real number *x*0 such that

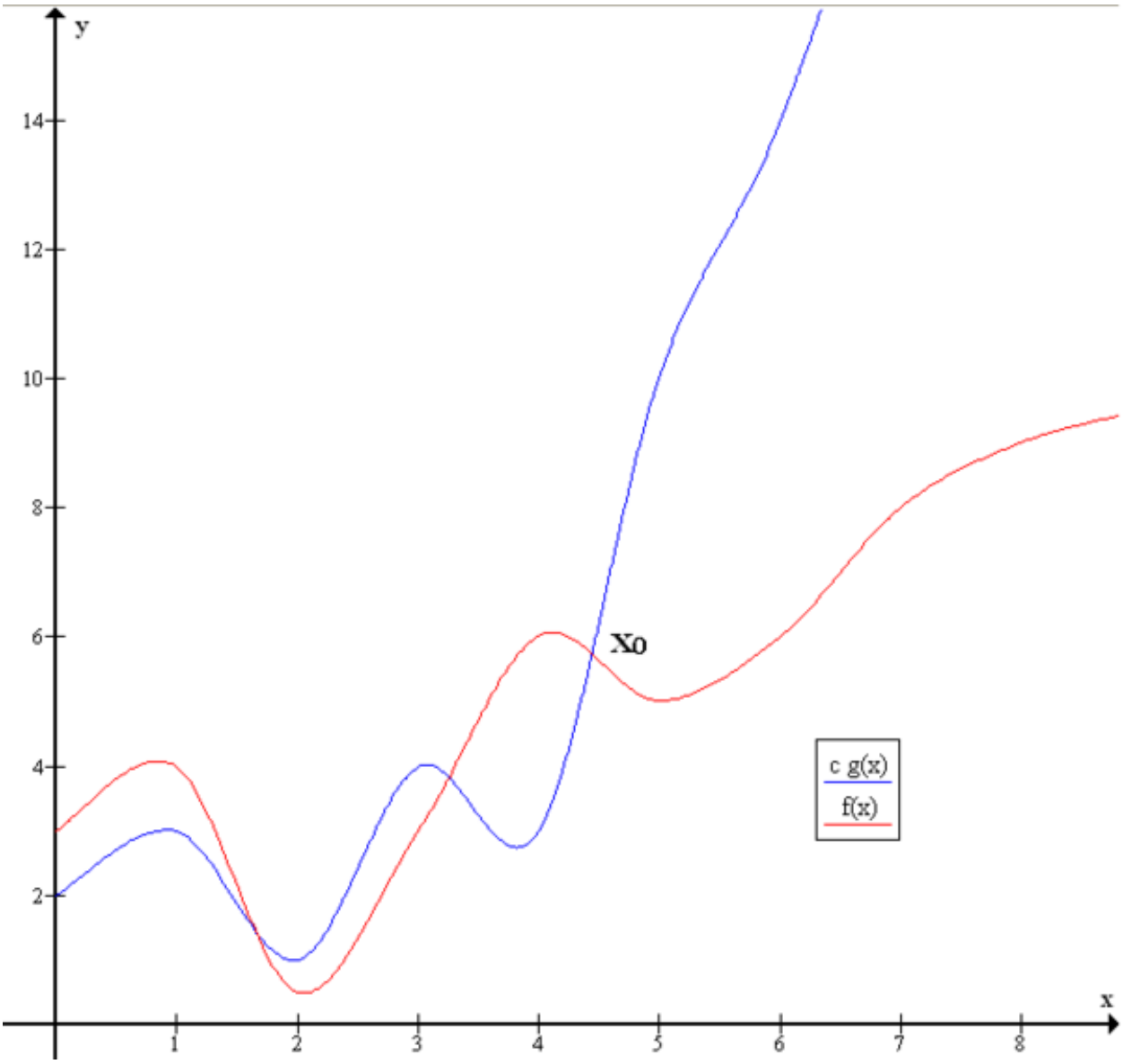
$$|f(x)|\leq M|g(x)| \text{ for all } x\geq x0|f(x)|\leq M|g(x)| \text{ for all } x\geq x0$$

.



http://www.wolframalpha.com/input/?i=intersect+of+x%5E3+and+x%5E2

If $f(x)$ = x^2 and $g(x)$ = x^3 then we need to find an M and x0 such that f(x)≤Mg(x) for all x≥x0. For M=1 and x0 greater than 1.0 this is true.



Example

In typical usage, the formal definition of *O* notation is not used directly; rather, the *O* notation for a function *f* is derived by the following simplification rules:

- If *f*(*x*) is a sum of several terms, if there is one with largest growth rate, it can be kept, and all others omitted.
- If *f*(*x*) is a product of several factors, any constants (terms in the product that do not depend on *x*) can be omitted.

For example, let *f*(*x*) = 6*x*^4^ − 2*x*^3^ + 5, and suppose we wish to simplify this function, using *O*-notation, to describe its growth rate as *x* approaches infinity. This function is the sum of three terms: 6*x*^4^, −2*x*^3^, and 5. Of these three terms, the one with the highest growth rate is the one with the largest exponent as a function of *x*, namely 6*x*^4^.

Question Arrange the following functions in increasing order of growth:

- log(33n)
- $33^n + 11^n$
- $0.1^n + 1$
- n / log(n)
- $\log(n) + n^{1/3}$
- log(log(n))
- $\log(n^3)$
- $n!\ e^n$
- √n
- 5^n

Solution:

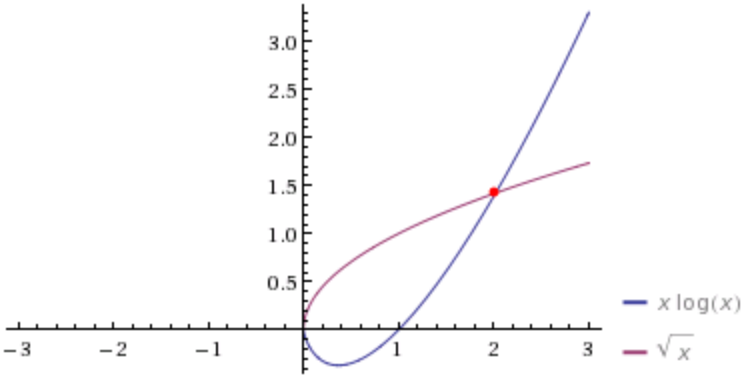
1. $0.1^n + 1$ Θ(1) (Exponentially decreasing)
2. log(log n) Θ(log(log n))
3. log 33n Θ(log n)
4. log (n^3) Θ(log n)
5. log n + $n^{1/3}$ Θ($n^{1/3}$)
6. √n Θ($n^{1/2}$)
7. n / log(n) Θ(n/log n)
8. 5^n Θ(5^n)
9. $33^n + 11^n$ Θ(33^n) (Exponentially increasing)
10. $n!\ e^n$ Θ(n(n+1/2))

Question

One algorithm requires n $\log_2(n)$ seconds and another algorithm requires √n seconds. Which one is asymptotically faster? What is the cross-over value of n? (The value at which the curves intersect?

Solution:

The first algorithm, n $\log_2(n)$, is asymptotically faster than the second, √n.



The order of growth of √n is less (slower) than the order of growth of n $\log_2$ n
Crossover point of n is 2.02075

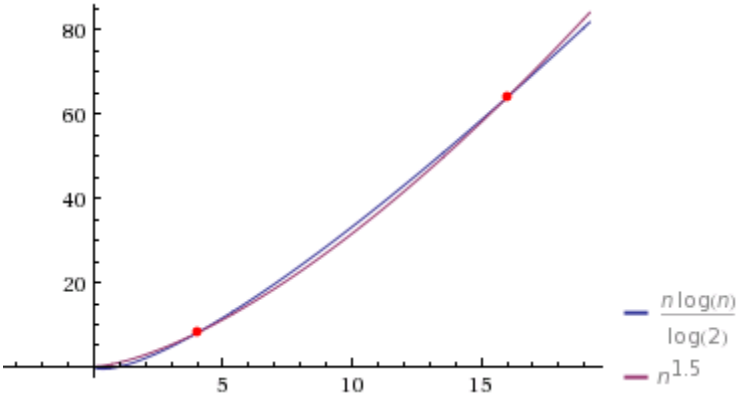
Question

Is n log(n) big-O of $n^{1.5}$? Prove your answer. The intersections of the two functions is

$n = 4$ and

$$n = e^{-2\,W_{-1}\left(-\frac{\log(2)}{2}\right)} \approx 16.0000$$

Solution:



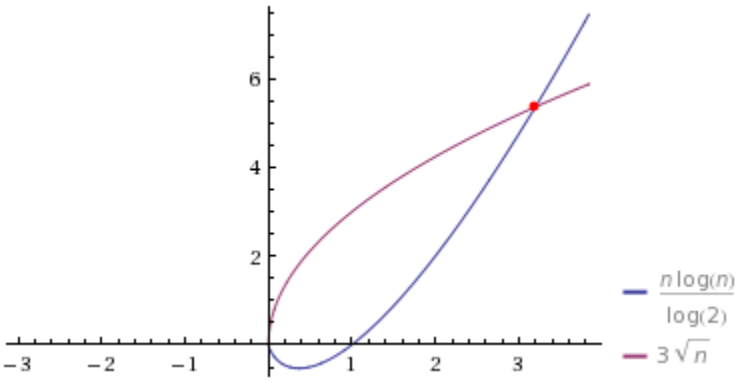
No. A function f is O(g) iff there exists a C and N such that f(x) ≤ C |g(x)| for all x ≥ N. In the case of proving big-O, you need to find the C and N. We can see after an N=16 and a C=1 the x of $n^{1.5}$ ≥ n log(n) as the n goes to infinity.

Question

Is n log(n) big-O of 3 √n? Prove your answer. The intersection of the two functions is

$$n = e^{2\,W\left(\frac{3\log(2)}{2}\right)} \approx 3.19857$$

Solution:



Yes. A function f is O(g) iff there exists a C and N such that f(x) ≤ C |g(x)| for all x ≥ N. In the case of proving big-O, you need to find the C and N. We can see after an N=5 and a C=1 the x of n log(n) ≥ 3 √n as the x goes to infinity.

Graphs (graph search)

In mathematics, and more specifically in graph theory, a graph is a structure amounting to a set of objects in which some pairs of the objects are in some sense "related". The objects correspond to mathematical abstractions called vertices (also called nodes or points) and each of the related pairs of vertices is called an edge (also called an arc or line). Typically, a graph is depicted in diagrammatic form as a set of dots for the vertices, joined by lines or curves for the edges. Graphs are one of the objects of study in discrete mathematics.

Undirected graph

An undirected graph is a graph in which edges have no orientation. The edge is identical to the edge , i.e., they are not ordered pairs, but sets } (or 2-multisets) of vertices. The maximum number of edges in an undirected graph without a loop is .

Directed graph

A directed graph or digraph is a graph in which edges have orientations. It is written as an ordered pair (sometimes) with

V a [set] whose [elements] are called vertices, nodes, or points;
A a set of [ordered pairs] of vertices, called arrows, directed edges (sometimes simply edges with the corresponding set named E instead of A), directed arcs, or directed lines.

An arrow is considered to be directed from x to y; y is called the head and x is called the tail of the arrow; y is said to be a direct successor of x and x is said to be a direct predecessor of y. If a [path] leads from x to y, then y is said to be a successor of x and reachable from x, and x is said to be a predecessor of y. The arrow is called the inverted arrow of .

A directed graph G is called symmetric if, for every arrow in G, the corresponding inverted arrow also belongs to G. A symmetric loopless directed graph is equivalent to a simple undirected graph , where

the pairs of inverse arrows in A correspond one-to-one with the edges in E; thus the number of edges in G' is , that is half the number of arrows in G.

i. Independent Set

"In graph theory, an independent set or stable set is a set of vertices in a graph, no two of which are adjacent. That is, it is a set I of vertices such that for every two vertices in I, there is no edge connecting the two." - Wikipedia

ii. Bipartite graph

"In the mathematical field of graph theory, a bipartite graph is a graph whose vertices can be divided into two disjoint sets and such that every edge connects a vertex in to one in . Vertex set and are often denoted as partite sets." - Wikipedia

iii. Spanning Tree

"In the mathematical field of graph theory, a spanning tree T of a connected, undirected graph G is a tree that includes all of the vertices and some or all of the edges of G." - Wikipedia

iv. Connected component

"In graph theory, a connected component of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph." - Wikipedia

i. Cut set

"In graph theory, a cut is a partition of the vertices of a graph into two disjoint subsets that are joined by at least one edge. The cut-set of the cut is the set of edges whose end points are in different subsets of the partition." - Wikipedia

i. Graph cycle

"A graph cycle consists of a sequence of vertices starting and ending at the same vertex, with each two consecutive vertices in the sequence adjacent to each other in the graph. The other type of cycle, sometimes called a simple cycle." " - Wikipedia

ii. Tree graph

"In graph theory, a tree is an undirected graph in which any two vertices are connected by exactly one simple path. In other words, any connected graph without simple cycles is a tree. " – Wikipedia

iii. BFS

"In graph theory, breadth-first search (BFS) is a strategy for searching in a graph when search is limited to essentially two operations: (a) visit and inspect a node of a graph; (b) gain access to visit the nodes that neighbor the currently visited node. The BFS begins at a root node and inspects all the neighboring nodes. Then for each of those neighbor nodes in turn, it inspects their neighbor nodes which were unvisited, and so on. The time complexity can be expressed as O(|V|+|E|) since every vertex and every edge will be explored in the worst case.:" - Wikipedia

iv. Greedy algorithm

"A greedy algorithm is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum." -Wikipedia

v. Adjacency List

"In graph theory and computer science, an adjacency list representation of a graph is a collection of unordered lists, one for each vertex in the graph. Each list describes the set of neighbors of its vertex." - Wikipedia

Greedy Algorithms (shortest paths and MSTs)

A greedy algorithm is an algorithmic paradigm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum. In many problems, a greedy strategy does not in general produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a global optimal solution in a reasonable time.

For example, a greedy strategy for the traveling salesman problem (which is of a high computational complexity) is the following heuristic: "At each stage visit an unvisited city nearest to the current city". This heuristic need not find a best solution, but terminates in a reasonable number of steps; finding an optimal solution typically requires unreasonably many steps.

Dijkstra's algorithm

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.

The algorithm exists in many variants; Dijkstra's original variant found the shortest path between two nodes, but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree.

Let the node at which we are starting be called the **initial node**. Let the **distance of node *Y*** be the distance from the **initial node** to *Y*. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.

1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
2. Set the initial node as current. Mark all other nodes unvisited. Create a set of all the unvisited nodes called the *unvisited set*.
3. For the current node, consider all of its neighbors and calculate their *tentative* distances. Compare the newly calculated *tentative* distance to the current assigned value and assign the smaller one. For example, if the current node *A* is marked with a distance of 6, and the edge connecting it with a neighbor *B* has length 2, then the distance to *B* (through *A*) will be 6 + 2 = 8. If B was previously marked with a distance greater than 8 then change it to 8. Otherwise, keep the current value.
4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the *unvisited set*. A visited node will never be checked again.

a. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the *unvisited set* is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.

b. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new “current node”, and go back to step 3.

Prim's algorithm

In computer science, Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

The algorithm may informally be described as performing the following steps:

In more detail, it may be implemented following the pseudocode below.

As described above, the starting vertex for the algorithm will be chosen arbitrarily, because the first iteration of the main loop of the algorithm will have a set of vertices in *Q* that all have equal weights, and the algorithm will automatically start a new tree in *F* when it completes a spanning tree of each connected component of the input graph. The algorithm may be modified to start with any particular vertex *s* by setting *C*[*s*] to be a number smaller than the other values of *C* (for instance, zero), and it may be modified to only find a single spanning tree rather than an entire spanning forest (matching more closely the informal description) by stopping whenever it encounters another vertex flagged as having no associated edge.

Different variations of the algorithm differ from each other in how the set *Q* is implemented: as a simple [linked list] or [array] of vertices, or as a more complicated [priority queue] data structure. This choice leads to differences in the [time complexity] of the algorithm. In general, a priority queue will be quicker at finding the vertex *v* with minimum cost, but will entail more expensive updates when the value of *C*[*w*] changes.

Kruskal's algorithm

Kruskal's algorithm is a minimum-spanning-tree algorithm which finds an edge of the least possible weight that connects any two trees in the forest. It is a greedy algorithm in graph theory as it finds a minimum spanning tree for a connected weighted graph adding increasing cost arcs at each step. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a minimum spanning forest (a minimum spanning tree for each connected component).

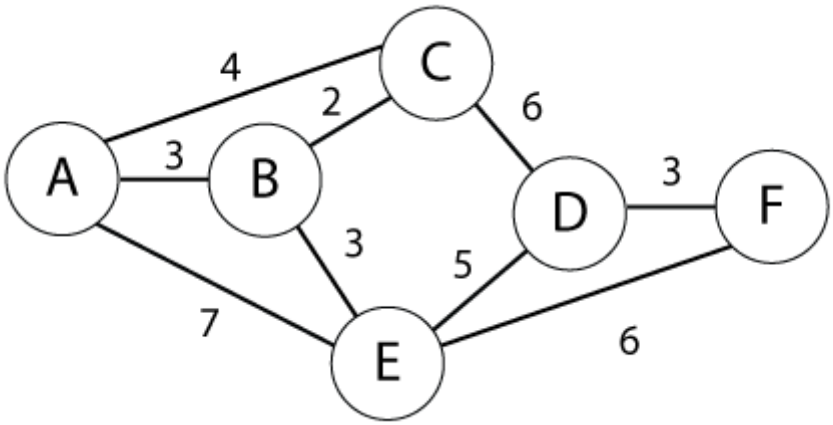
Algorithm

- create a graph *F* (a set of trees), where each vertex in the graph is a separate [tree]
- create a set *S* containing all the edges in the graph
- while *S* is [nonempty] and *F* is not yet [spanning]
    - remove an edge with minimum weight from *S*
    - if the removed edge connects two different trees then add it to the forest *F*, combining two trees into a single tree

At the termination of the algorithm, the forest forms a minimum spanning forest of the graph. If the graph is connected, the forest has a single component and forms a minimum spanning tree

Question

Use Kruskal's algorithm to find a minimum spanning tree for the connected weighted graph below:



What is the Time Complexity of Kruskal's algorithm?

Solution:

Kruskal's algorithm pseudocode:

A. Create a forest T (a set of trees), where each vertex in the graph is a separate tree
B. Create a set S containing all the edges in the graph
C. Sort edges by weight

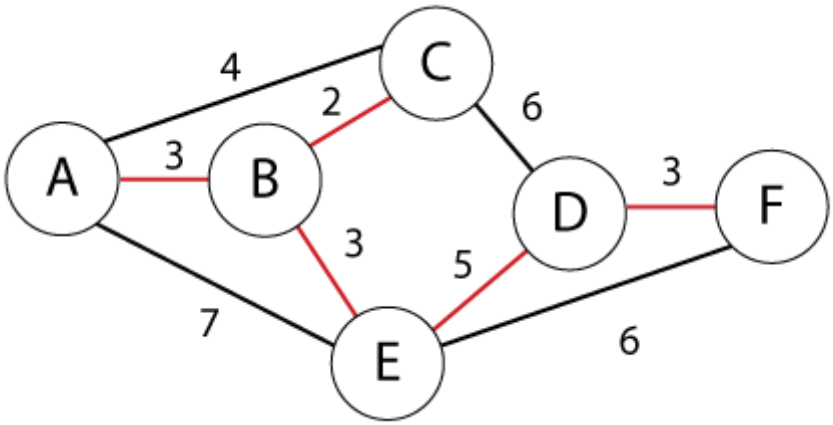
While S is nonempty and T is not yet spanning:

I. remove an edge with minimum weight from S
II. if that edge connects two different trees, then add it to the forest, combining two trees into a single tree, otherwise discard that edge.

Steps:

I. Connect B-C (2)
II. Connect A-B (3)
III. Connect D-F (3)
IV. Connect B-E (3)
V. Skip A-C (4) (Forms cycle)
VI. Connect E-D (4)

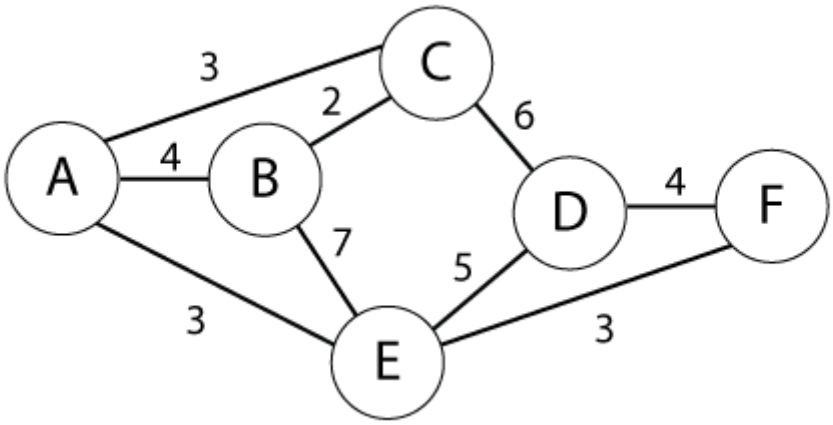
Stop. MST formed (5 edges, 6 vertices)



MST = {A-B , B-C , B-E, E-D, D-F}
Kruskal's algorithm is O(E log V) time. Where E is the set of edges and V is the set of vertices.

Question

Use Prim's algorithm to find a minimum spanning tree for the connected weighted graph below. *Show your work.*



What is the Time Complexity of Prim's algorithm?

Solution:

*Note: Using Kruskal's algorithm is NOT correct. It says: "*Use Prim's algorithm to find a minimum spanning tree*".*

Prim's algorithm (Queue edges for Minimum Spanning Tree) – is a greedy algorithm that finds a minimum spanning tree for a connected weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. Runs in $O(n^2)$ with an array; O(m log n) with a binary heap

Step 1: Choose any starting vertex. Look at all edges connecting to the vertex and choose the one with the lowest weight and add this to the tree.

Step 2: Look at all edges connected to the tree. Choose the one with the lowest weight and add to the tree.

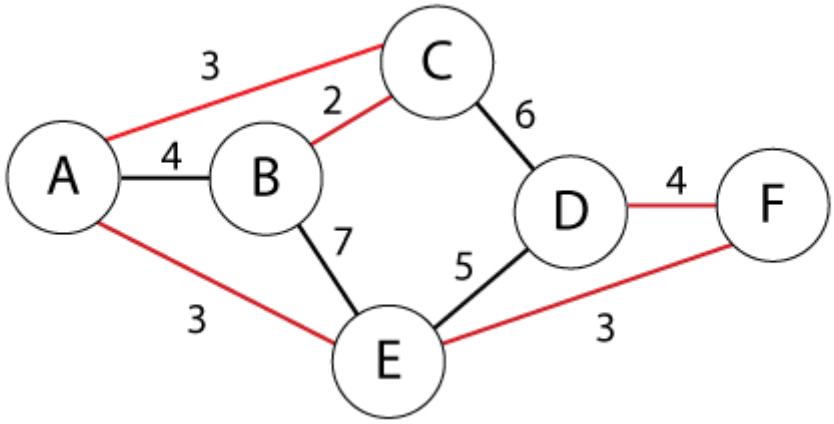
Step 3: Repeat step 2 until all vertices are in the tree (n-1 edges).

```
Prim(G, c) {
        foreach (v in V) a[v] <- ∞
         Initialize an empty priority queue Q
        foreach (v in V) insert v onto Q
        Initialize set of explored nodes S = {}

        while (Q is not empty) {
                u =delete min element from Q
                S =S (union symbol){u}
                foreach (edge e = (u, v) incident to u)
                if ((v (is not an element symbol) (S) and (ce <a[v]))
                        decrease priority a[v] to ce
}
```

In other words

Take the minimum edge of the cut-set each time.

0: A S = {A}
1: A-C (3) or A-E (3) is min-cut take A-E S = {A,E}
2: E-F (3) or A-C (3) is min-cut take E-F S = {A,E,F}
3: A-C (3) is min-cut take A-C S = {A,E,F,C}
4: B-C (2) is min-cut take B-C S = {A,E,F,C,B}
5: D-F (4) is min-cut take D-F S = {A,E,F,C,B,D}
Done n-1 edges. (5)



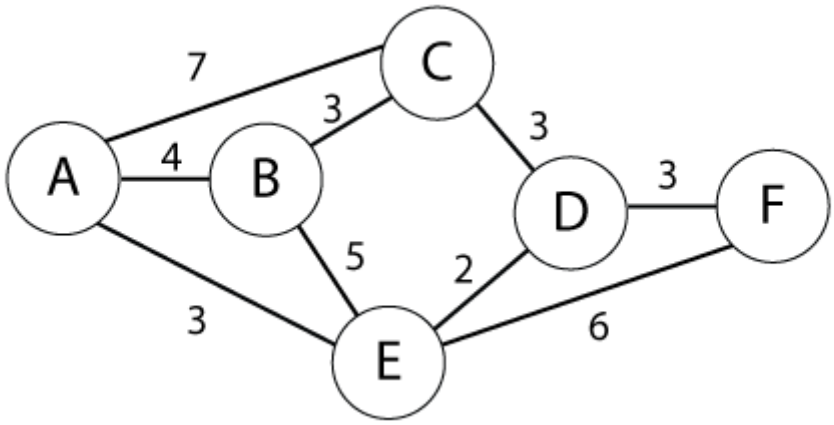
MST = {A-E, E-F, A-C, C-B, D-F}
MST weight =3+3+3+2+4=15

Prim's algorithm time depends on the implementation. Any of the below are accepted. Where E is the set of edges and V is the set of vertices.

Minimum edge weight data structure	Time complexity (total)
adjacency matrix, searching	$O(\|V\|^2)$
binary heap and adjacency list	$O((\|V\| + \|E\|) \log \|V\|) = O(\|E\| \log \|V\|)$
Fibonacci heap and adjacency list	$O(\|E\| + \|V\| \log \|V\|)$

Question

Find shortest path from A to F in the graph below using Dijkstra's algorithm. *Show your steps.*



Solution:

Given a graph, G, with edges E of the form (v1, v2) and vertices V, and a source vertex, s

dist : array of distances from the source to each vertex
prev : array of pointers to preceding vertices
i : loop index

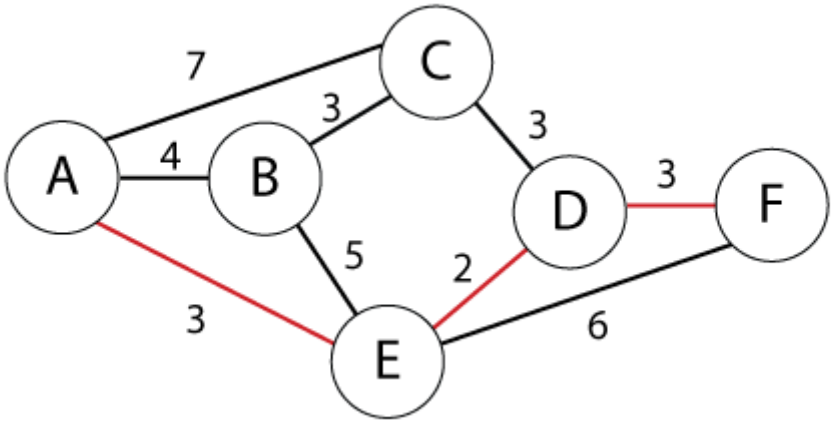
```
F    : list of finished vertices
U    : list or heap unfinished vertices

/* Initialization: set every distance to INFINITY until we discover a path */
for i = 0 to |V| - 1
    dist[i] = INFINITY
    prev[i] = NULL
end
```

```
while(F is missing a vertex)
   pick the vertex, v, in U with the shortest path to s
   add v to F
   for each edge of v, (v1, v2)
       /* The next step is sometimes given the confusing name "relaxation"
       if(dist[v1] + length(v1, v2) < dist[v2])
          dist[v2] = dist[v1] + length(v1, v2)
          prev[v2] = v1
          possibly update U, depending on implementation
       end if
    end for
end while
```

Source A

		A	B	C	D	E	F
1: A {A}	A	0	(4,A)	(7,A)	INF	(3, A)**	INF
2: E {A,E}	E	0	(4,A)**	(7,A)	(5,E)	(3, A)	(9,E)
3: B {A,E,B}	B	0	(4,A)	(7,A)	(5,E)**	(3, A)	(9,E)
4: D {A,E,B,D}	D	0	(4,A)	(7,A) **	(5,E)	(3, A)	(8,D)
5: C {A,E,B,D,C}	C	0	(4,A)	(7,A)	(5,E)	(3, A)	(8,D) **
4: F {A,E,B,D,C,F}	F	0	(4,A)	(7,A)	(5,E)	(3, A)	(8,D)

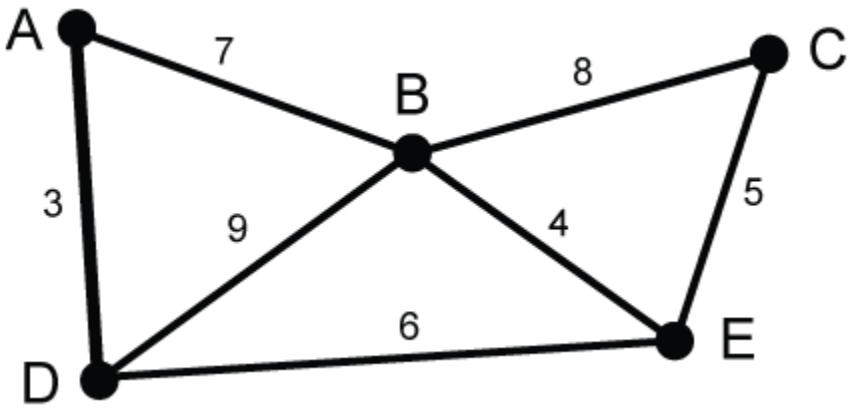
** U with the shortest path to s
Red indicates a vertex has been moved from U to F



Now we return our final shortest path, which is: A → E → D → F Cost (3+2+3 = 8)

Question

Use Kruskal's algorithm to find a minimum spanning tree for the connected weighted graph below:



What is the Time Complexity of Kruskal's algorithm?

Kruskal's algorithm pseudocode:

A. Create a forest T (a set of trees), where each vertex in the graph is a separate tree
B. Create a set S containing all the edges in the graph
C. Sort edges by weight

While S is nonempty and T is not yet spanning:
III. remove an edge with minimum weight from S
IV. if that edge connects two different trees, then add it to the forest, combining two trees into a single tree, otherwise discard that edge.

Solution:

Steps:

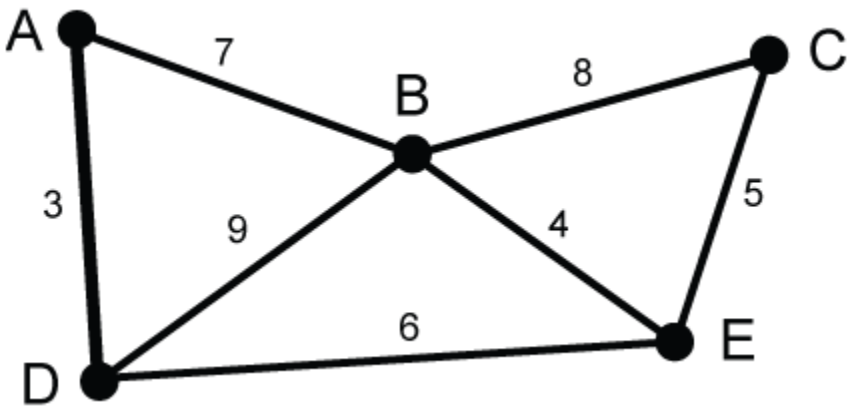
VII. Connect A-D (3)
VIII. Connect B-E (4)
IX. Connect C-E (5)
X. Connect D-E (6)

Stop. MST formed (4 edges, 5 vertices)

Kruskal's algorithm is O(E log V) time. Where E is the set of edges and V is the set of vertices.

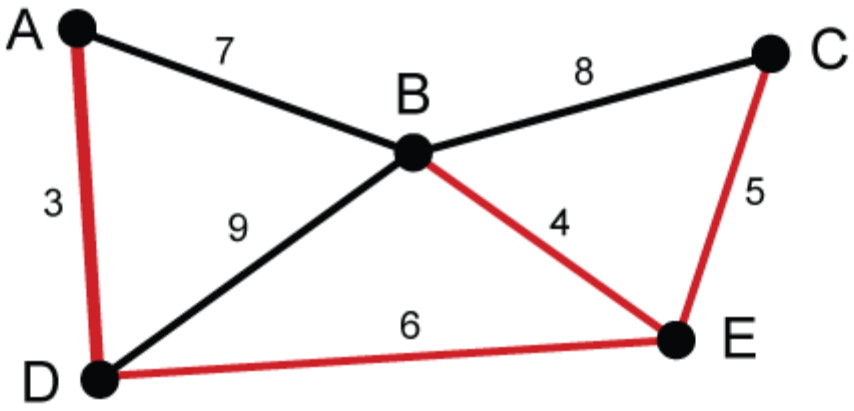
Question

Use Prim's algorithm to find a minimum spanning tree for the connected weighted graph below. *Show your work.*



What is the Time Complexity of Prim's algorithm?

Solution:



*Note: Using Kruskal's algorithm is NOT correct. It says: "*Use Prim's algorithm to find a minimum spanning tree*".*
Prim's algorithm (Queue edges for Minimum Spanning Tree) – is a greedy algorithm that finds a minimum spanning tree for a connected weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. Runs in $O(n^2)$ with an array; O(m log n) with a binary heap

Step 1: Choose any starting vertex. Look at all edges connecting to the vertex and choose the one with the lowest weight and add this to the tree.
Step 2: Look at all edges connected to the tree. Choose the one with the lowest weight and add to the tree.
Step 3: Repeat step 2 until all vertices are in the tree (n-1 edges).

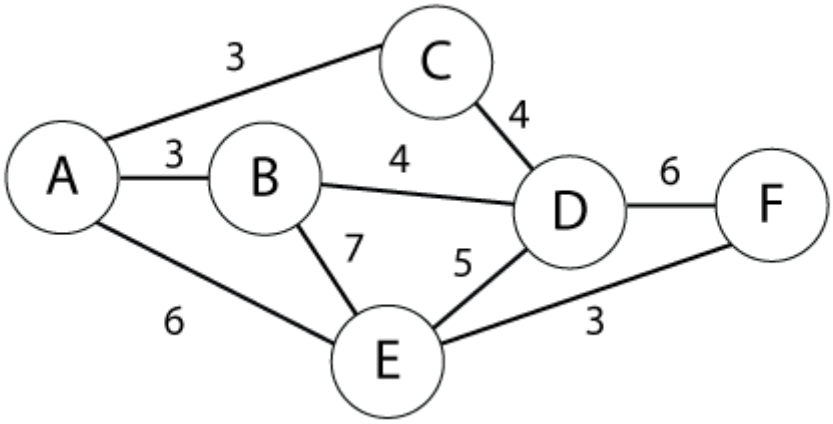
```
Prim(G, c) {
        foreach (v in V) a[v] <- ∞
         Initialize an empty priority queue Q
        foreach (v in V) insert v onto Q
        Initialize set of explored nodes S = {}

        while (Q is not empty) {
                u =delete min element from Q
                S =S (union symbol){u}
                foreach (edge e = (u, v) incident to u)
                if ((v (is not an element symbol) (S) and (ce <a[v]))
                        decrease priority a[v] to ce
}
```

In other words
Take the minimum edge of the cut-set each time.

0: A S = {A}
1: A-D < A-B take A-D S = {A,D}
2: D-E < A-B or D-B take D-E S = {A,D,E}
3: B-E < A-B or D-B or C-E take B-E S = {A,D,E,B}
4: C-E < A-B or D-B or B-C- take C-E S = {A,D,E,B,C}
Done n-1 edges.
MST = { A-D , D-E , B-E, C-E }

Question

Use Kruskal's algorithm to find a minimum spanning tree for the connected weighted graph below:



What is the Time Complexity of Kruskal's algorithm?

Solution:

Kruskal's algorithm pseudocode:

A. Create a forest T (a set of trees), where each vertex in the graph is a separate tree
B. Create a set S containing all the edges in the graph
C. Sort edges by weight

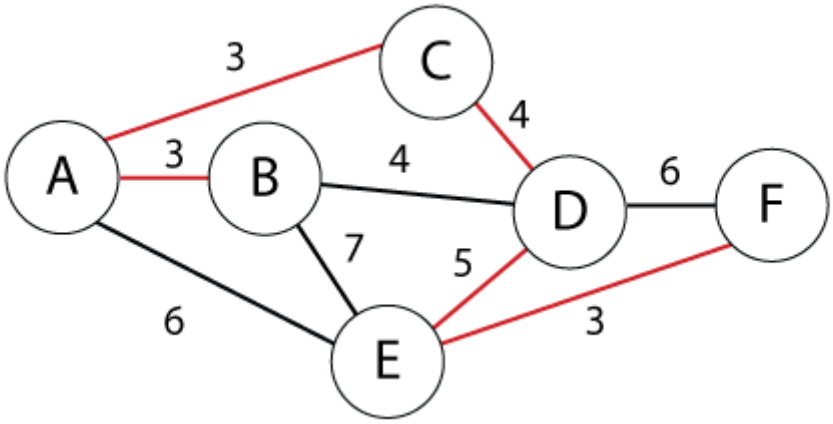
While S is nonempty and T is not yet spanning:
V. remove an edge with minimum weight from S
VI. if that edge connects two different trees, then add it to the forest, combining two trees into a single tree, otherwise discard that edge.

Steps:

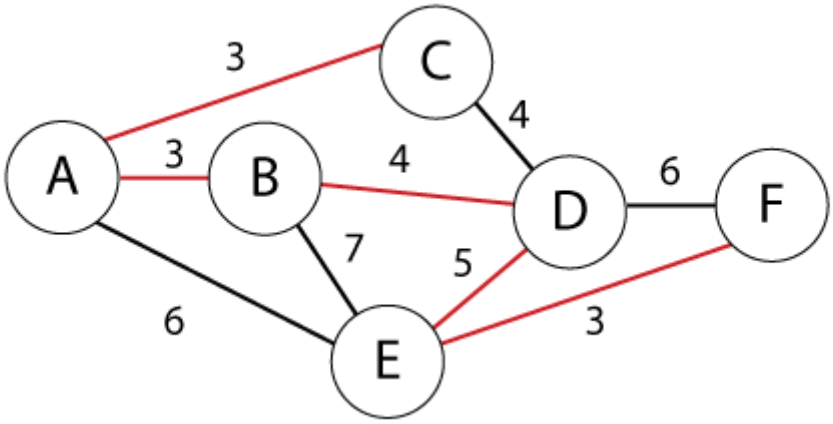
XI. Connect A-B (3)
XII. Connect A-C (3)
XIII. Connect E-F (3)
XIV. Connect C-D (4)
XV. Skip B-D cycle (4) (note: B-D could be connected and C-D skipped)
XVI. Connect D-E (5)

Stop. MST formed (5 edges, 6 vertices)

MST = {A-B, A-C, E-F, C-D, D-E} or {A-B, A-C, E-F, B-D, D-E}
MST weight = 3+3+3+4+5=18



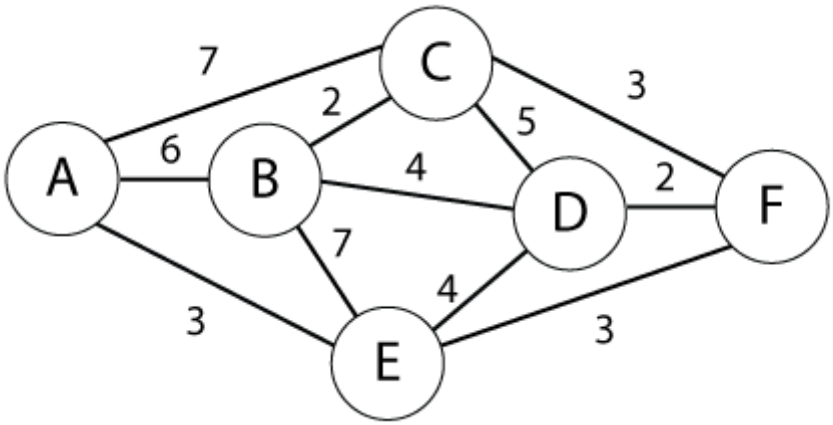
or



Kruskal's algorithm is O(E log V) time. Where E is the set of edges and V is the set of vertices.

Question

Use Prim's algorithm to find a minimum spanning tree for the connected weighted graph below. *Show your work.*



What is the Time Complexity of Prim's algorithm?

Solution:

*Note: Using Kruskal's algorithm is NOT correct. It says: "*Use Prim's algorithm to find a minimum spanning tree*".*

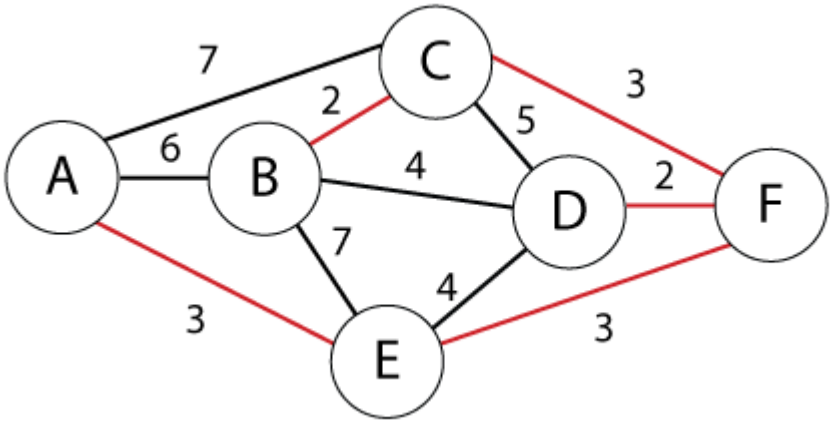
Step 1: Choose any starting vertex. Look at all edges connecting to the vertex and choose the one with the lowest weight and add this to the tree.
Step 2: Look at all edges connected to the tree. Choose the one with the lowest weight and add to the tree.
Step 3: Repeat step 2 until all vertices are in the tree (n-1 edges).

```
Prim(G, c) {
        foreach (v in V) a[v] <- ∞
         Initialize an empty priority queue Q
        foreach (v in V) insert v onto Q
        Initialize set of explored nodes S = {}

        while (Q is not empty) {
                u =delete min element from Q
                S =S (union symbol){u}
                foreach (edge e = (u, v) incident to u)
                if ((v (is not an element symbol) (S) and (ce <a[v]))
                        decrease priority a[v] to ce
}
```

In other words
Take the minimum edge of the cut-set each time.
0: A S = {A}
1: A-E (3) is min-cut take A-E S = {A,E}
2: E-F (3) is min-cut take E-F S = {A,E,F}
3: D-F (2) is min-cut take D-F S = {A,E,F,D}
4: C-F (3) is min-cut take C-F S = {A,E,F,D,C}

5: C-B (2) is min-cut take C-B S = {A,E,F,D,C,B}
Done n-1 edges.

MST = {A-E, E-F, D-F, C-F, C-B



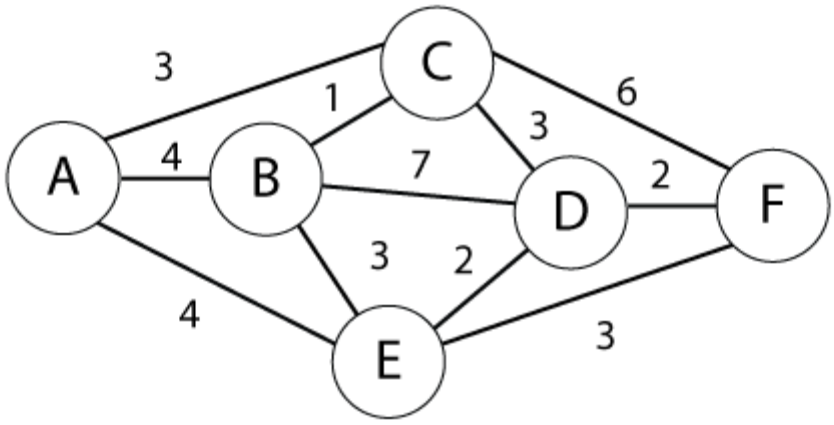
}

MST weight = 3+3+2+3+2=13
Prim's algorithm time depends on the implementation. Any of the below are accepted. Where E is the set of edges and V is the set of vertices.

Minimum edge weight data structure	**Time complexity (total)**
adjacency matrix, searching	$O(\|V\|^2)$
binary heap and adjacency list	$O((\|V\| + \|E\|) \log \|V\|) = O(\|E\| \log \|V\|)$
Fibonacci heap and adjacency list	$O(\|E\| + \|V\| \log \|V\|)$

Question

Find shortest path from A to F in the graph below using Dijkstra's algorithm. *Show your steps.*



Solution:

```
Given a graph, G, with edges E of the form (v1, v2) and vertices V, and a
source vertex, s

dist : array of distances from the source to each vertex
prev : array of pointers to preceding vertices
i    : loop index
F    : list of finished vertices
U    : list or heap unfinished vertices

/* Initialization: set every distance to INFINITY until we discover a path */
for i = 0 to |V| - 1
   dist[i] = INFINITY
   prev[i] = NULL
end
```

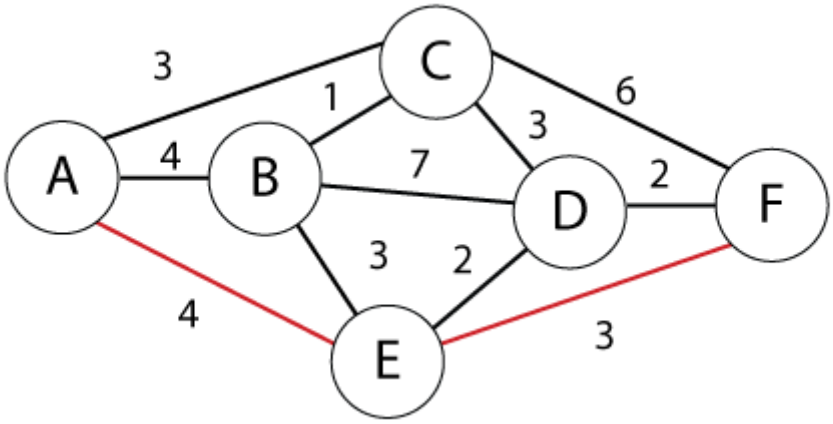
```
while(F is missing a vertex)
  pick the vertex, v, in U with the shortest path to s
  add v to F
  for each edge of v, (v1, v2)
      /* The next step is sometimes given the confusing name "relaxation"
      if(dist[v1] + length(v1, v2) < dist[v2])
         dist[v2] = dist[v1] + length(v1, v2)
         prev[v2] = v1
         possibly update U, depending on implementation
```

```
      end if
    end for
end while
```

Source A

		A	B	C	D	E	F
1: A {A}	A	0	(4,A)	(3,A)**	INF	(4, A)	INF
2: C {A,C}	C	0	(4,A)	(3,A)	(6,C)	(4, A)**	(9,C)
3: E {A,C,E}	E	0	(4,A) **	(3,A)	(6,C)	(4, A)	(7,E)
4: B {A,C,E,B}	B	0	(4,A)	(3,A)	(6,C) **	(4, A)	(7,E)
5: C {A,C,E,B,D}	D	0	(4,A)	(3,A)	(6,C)	(4, A)	(7,E) **
4: F {A,C,E,B,D,F}	F	0	(4,A)	(3,A)	(6,C)	(4, A)	(7,E)

** U with the shortest path to s
Note: Can take in A, C, E, B, D, F or A, C, B, E, D, F as E and B both have shortest path cost 4.
Red indicates a vertex has been moved from U to F



Now we return our final shortest path, which is: A → E → F Cost (4+3 = 7)

Divide and Conquer (sorting and selection)

In computer science, divide and conquer (D&C) is an algorithm design paradigm based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

The master theorem sometimes yields asymptotically tight bounds to some recurrences from [divide and conquer algorithms] that partition an input into smaller sub-problems of equal sizes, solve the sub-problems recursively, and then combine the sub-problem solutions to give a solution to the original problem. The time for such an algorithm can be expressed by adding the work that they perform at the top level of their recursion (to divide the problems into sub-problems and then combine the sub-problem solutions) together with the time made in the recursive calls of the algorithm.
If T(n)T(n) denotes the total time for the algorithm on an input of size n, and f(n)f(n) denotes the amount of time taken at the top level of the recurrence then the time can be expressed by a [recurrence relation] that takes the form:

$$T(n)=aT(nb)+f(n)T(n)=aT(nb)+f(n)$$

Here n is the size of an input problem, aa is the number of sub-problems in the recursion, and bb is the factor by which the sub-problem size is reduced in each recursive call. The theorem below also assumes that, as a base case for the recurrence, T(n)=Θ(1)T(n)=Θ(1) when n is less than some bound κ>0κ>0, the smallest input size that will lead to a recursive call.

Recurrences of this form often satisfy one of the three following regimes, based on how the work to split/recombine the problem f(x)f(x) relates to the *critical exponent* ccrit=logbaccrit=logb⁡a.

Question

For each of the following recurrences, give an expression for the runtime T(n) if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

i. $T(n) = 4T(n/2) + n^3 \log n$

ii. $T(n) = 2T(n/2) + n^{0.55}$

iii. $T(n) = 0.1^n T(n/2) + \log n$

iv. $T(n) = 9T(n/3) - n^2$

v. $T(n) = n^2 T(n/2) + n$

Solutions:

i. $T(n) = 4T(n/2) + n^2 \log n$ Case 3:- $T(n) = \text{Theta}(n^2 \log n)$

A = 4, B= 2, k=2, $f(n) = n^2 \log n$
Case 2: p=0
F(n) = Θ (n^2log0n)
T(n) = Θ (n^2log1n)

ii. $T(n) = 2T(n/2) + n^{0.33}$ Case 1: - T(n) = Theta(n)

A = 2, B = 2, k = 1, $f(n) = n^{0.33}$
Case 1: ε = .1
$T(n) = \Theta(n^k) = \Theta(n)$

iii. $T(n) = 0.1^n T(n/2) + \log n$ it does not apply :- a < 1

iv. $T(n) = 9T(n/3) - n^2$ it does not apply :- f(n) is negative

v. T(n) = 2T (n/2) + n/ log n it does not apply :- Does not apply, non-polynomial difference between f(n) and n^k

Question

Part A. Consider the modified binary search algorithm so that it splits the input not into two sets of almost-equal sizes, but into three sets of sizes approximately one-third. Write down the recurrence for this ternary search algorithm and find the asymptotic complexity of this algorithm.

Part B. Consider another variation of the binary search algorithm so that it splits the input not only into two sets of almost equal sizes, but into two sets of sizes approximately one-third and two-thirds. Write down the recurrence for this search algorithm and find the asymptotic complexity of this algorithm.

Solution:

The recurrence for normal binary search is $T(n) = T(n/2)+c$. This accounts for one comparison and then recursively call on the appropriate partition. For ternary search we make two comparisons which partitions the list into three sections roughly n/3 elements and recurs on the appropriate partition. Thus analogously, the recurrence for the number of comparisons made by this ternary search is: $T(n) = T(n /3) + c$. However, just as for binary search the second case of the Master Theorem applies. ($\log_b a = c = 0$ so they are the same thus case 2)

We therefore conclude that $T(n) \in \Theta(\log(n))$.

We now consider a slightly modified take on ternary search in which only one comparison is made which creates two partitions, one of roughly n/3 elements and the other of 2n/3. Here the worst case arises when the recursive call is on the larger 2n/3-element partition. Thus the recurrence corresponding to this worst case number of comparisons is
$T(n) = T(2n/3) + c$.

but again the second case of the Master Theorem applies (($\log_b a = c = 0$ so they are the same thus case 2) placing $T(n) \in \Theta(\log n)$.

Since b = 3/2 This function is $\Theta(\log_{3/2} n)$ but $\Theta(\log n)$ is fine.

The best case is exactly the same (and thus so must also be the average). It is interesting to note that we will get the same results for general k-ary search (as long as k is a fixed constant which does not depend on n) as n approaches infinity. The recurrence becomes one of
$T(n) = T(n/k) + k - 1$
$T(n) = T((k-1)n/k)+1$

depending on whether we are talking about part (a) or part (b). For each recurrence case two of the Master Theorem applies.

Question

Sort the list of integers below using Quicksort. Show your work. What is the worst case and average case performance (big-O) of Quicksort? Write a worst case and average case recurrence relation for Quicksort.

(22, 13, 26, 1, 12, 27, 33, 15)

Solution:

Choose pivot as n-1 item. Use that to partition the array in-place.

```
Quicksort(A as array, low as int, high as int)
   if (low < high)
      pivot_location = Partition(A,low,high)
   Quicksort(A,low, pivot_location - 1)
   Quicksort(A, pivot_location + 1, high)

Partition(A as array, low as int, high as int)
    pivot = A[low]
    leftwall = low

    for i = low + 1 to high
       if (A[i] < pivot) then
          leftwall = leftwall + 1
          swap(A[i], A[leftwall])

    swap(A[low],A[leftwall])

   return (leftwall)
```

Assume the Pivot is last element of array
Assume we are swapping in-place around the pivot

Start: (22, 13, 26, 1, 12, 27, 33, 15)

Next: QS(22, 13, 26, 1, 12, 27, 33, 15)
Next: QS(1,13,12)+QS(15)+QS(22,27,33,26)
Next: QS(1,12)+QS(13)+QS(15)+QS(22)+QS(26,33,27)
Next: QS(1)+QS(12)+QS(13)+QS(15)+QS(22)+QS(26,27)+QS(33)

Next: QS(1)+QS(12)+QS(13)+QS(15)+QS(22)+QS(26)+QS(27)+QS(33)

Sorted: 1, 12, 13, 15, 22, 26, 27, 33
Worst case performance (big-O): $O(n^2)$

Average case performance (big-O): O(n log n)

Worst case Recurrence relation: T(n) = T(0) + T(n-1) + O(n) = T(n-1) + O(n)

Average recurrence relation: T(n) = 2 T(n/2) + O(n)

Question

Master Theorem: For the following recurrence, give an expression for the runtime T(n) if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

T(n) = 8T (n/2)+ n

Solution:

i. T(n) = 8T (n/2)+ n

$k = \log_b a = \log_2 8 = \log(8)/\log(2) = 3$
f(n) = n which is θ(n)

Case 1 So $\theta(n^k)$ or $\theta(n^{\log_b a})$
Therefore the runtime T(n) is $\theta(n^3)$

Question

Master Theorem: For the following recurrence, give an expression for the runtime T(n) if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

$T(n) = n^2T (n/2) + \log n$

Solution:

Does not apply (a is not constant)

Question

Master Theorem: For the following recurrence, give an expression for the runtime T(n) if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

T(n) = 17T (n/3)− n3

Solution:

Does not apply (f(n) is not positive)

Question

Sort the list of integers below using Merge sort. Show your work. What is the Worst case performance (big-O) of Merge sort? Write a recurrence relation for Merge sort.

(22, 13, 26, 1, 12, 27, 33, 15)

Solution:

Original	**22**	**13**	**26**	**1**	**12**	**27**	**33**	**15**							
Divide in 2	**22**	**13**	**26**	**1**		**12**	**27**	**33**	**15**						
Divide in 4	**22**	**13**		**26**	**1**		**12**	**27**		**33**	**15**				
Divide in 8	**22**		**13**		**26**		**1**		**12**		**27**		**33**		**15**
Merge 1	**13**	**22**			**1**	**26**			**12**	**27**			**15**	**33**	
Merge 2	**1**	**13**	**22**	**26**					**12**	**15**	**27**	**33**			
Merge 3	**1**	**12**	**13**	**15**	**22**	**26**	**27**	**33**							

Worst case performance (big-O): O(n log n)

Recurrence relation: T(n) = 2 T(n/2) + n

Dynamic Programming (basic techniques. sequence alignment, Bellman-Ford)

In computer science, mathematics, management science, economics and bioinformatics, dynamic programming (also known as dynamic optimization) is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions. The next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed solution, thereby saving computation time at the expense of a (hopefully) modest expenditure in storage space. (Each of the subproblem solutions is indexed in some way, typically based on the values of its input parameters, so as to facilitate its lookup.) The technique of storing solutions to subproblems instead of recomputing them is called "memoization".

Dynamic programming algorithms are often used for optimization. A dynamic programming algorithm will examine the previously solved subproblems and will combine their solutions to give the best solution for the given problem. In comparison, a greedy algorithm treats the solution as some sequence of steps and picks the locally optimal choice at each step. Using a greedy algorithm does not guarantee an optimal solution, because picking locally optimal choices may result in a bad global solution, but it is often faster to calculate.

Bellman-Ford algorithm

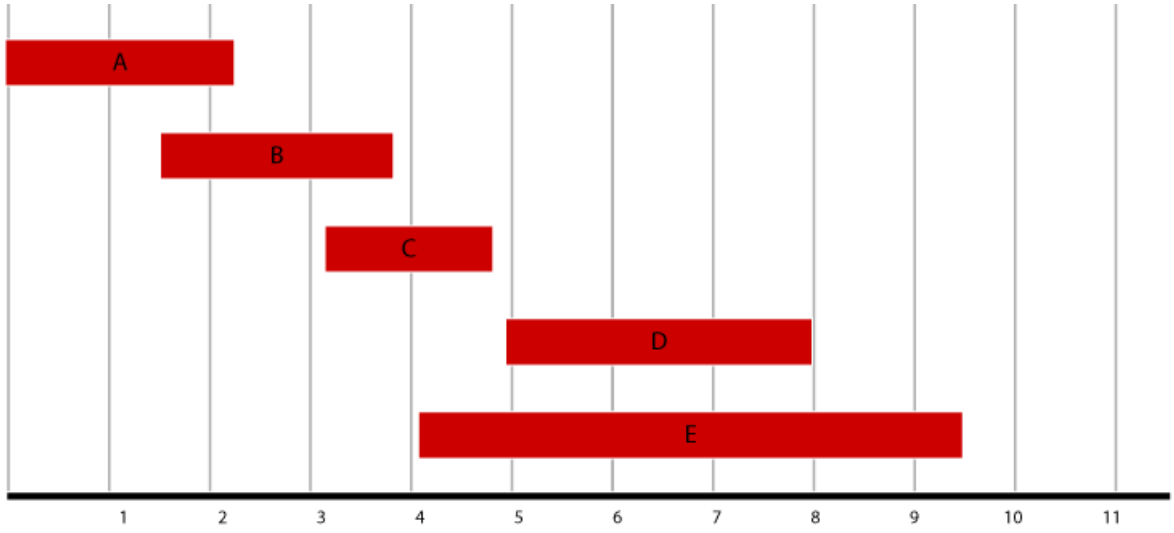
Like Dijkstra's Algorithm, Bellman–Ford is based on the principle of [relaxation], in which an approximation to the correct distance is gradually replaced by more accurate values until eventually reaching the optimum solution. In both algorithms, the approximate distance to each vertex is always an overestimate of the true distance, and is replaced by the minimum of its old value with the length of a newly found path. However, Dijkstra's algorithm uses a priority queue to [greedily] select the closest vertex that has not yet been processed, and performs this relaxation process on all of its outgoing edges; by contrast, the Bellman–Ford algorithm simply relaxes *all* the edges, and does this |V|−1 times, where |V is the number of vertices in the graph. In each of these repetitions, the number of vertices with correctly calculated distances grows, from which it follows that eventually all vertices will have their correct distances. This method allows the Bellman–Ford algorithm to be applied to a wider class of inputs than Dijkstra.\

Bellman–Ford runs in O(|V| · |E|)O(|V| · |E|) [time], where |V|and |E| are the number of vertices and edges respectively.

Simply put, the algorithm initializes the distance to the source to 0 and all other nodes to infinity. Then for all edges, if the distance to the destination can be shortened by taking the edge, the distance is updated to the new lower value. At each iteration ii that the edges are scanned, the algorithm finds all shortest paths of at most length ii edges (and possibly some paths longer than iedges). Since the longest possible path without a cycle can be |V|−1 edges, the edges must be scanned |V|−1| times to ensure the shortest path has been found for all nodes. A final scan of all the edges is performed and if any distance is updated, then a path of length |V| edges has been found which can only occur if at least one negative cycle exists in the graph.

Question

Given the five intervals below, and their associated values; select a subset of non-overlapping intervals with the maximum combined value. Use dynamic programming. *Show your work.*



Interval	Value
A	3
B	4
C	5
D	2
E	4

Solutions:

Interval	Value	Previous	Max
A	5	n/a	Max(5, 0) = 5
B	4	n/a	Max(4, 5) = 5
C	1	A	Max(5, 3+5) = 8
D	2	C	Max(8, 2+8) = 10
E	4	B	Max(10, 5+4) = 10

Interval	Trace(i)	S
E	4 + 5 < 10	{}
D	2+8=10, jump to C	{D}

C	5+3=8, jump to A	{D,C}
B	jump to A	
A	5 = 5	{D,C,A}

S = {A,C,D}

Question

Given the weights and values of the five items in the table below, select a subset of items with the maximum combined value that will fit in a knapsack with a weight limit, *W,* of 9. Use dynamic programming. *Show your work.*

Item $_i$	Value v_i	Weight w_i
1	2	3
2	3	4
3	4	2
4	3	2
5	5	5

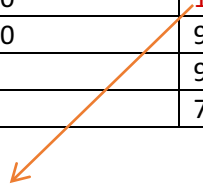
Capacity of knapsack W=9

Solution:

Capacity of knapsack W=9
Algorithm: given two arrays w[3, 4, 2, 2, 5] and v[3, 2, 4, 1, 5] and

for $I \leftarrow 1$ to n:
    for $x \leftarrow 1$ to w:
        if $w[I] > x$:
            $OPT[I, x] \leftarrow OPT[i - 1, x]$
        else
            $OPT[I, x] \leftarrow \max(OPT[i - 1, x]; OPT[i - 1, x - w[i] + v[i])$

9	2	5	9	10	12 <--
8	2	5	7	10	9
7	2	5	7	9	9
6	2	3	7	7	7

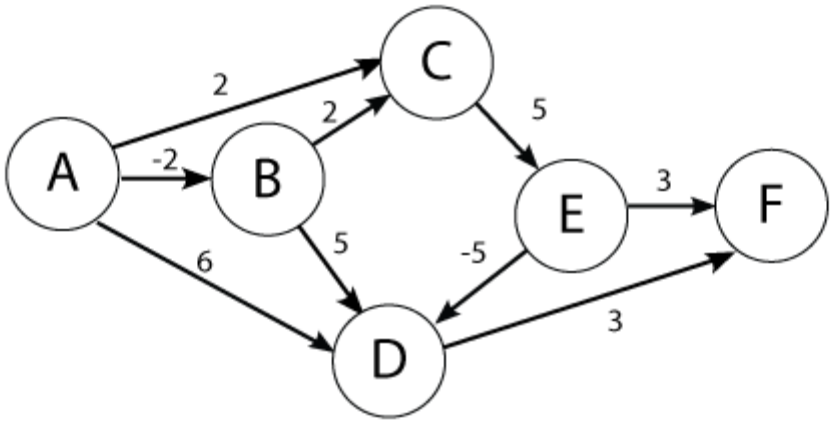


5	2	3	6	7	7
4	2	3	4	7 <--	7
3	2	2	4	4	4
2	0	0	4 <--	4	4
1	0	0	0	0	0
	1	**2**	**3**	**4**	**5**

We used items 3, 4 and 5 for a combined value of 12 in the knapsack.

S={3,4,5}

Question



Use the Bellman-Ford algorithm to find the shortest path from node A to F in the weighted directed graph above. *Show your work.*

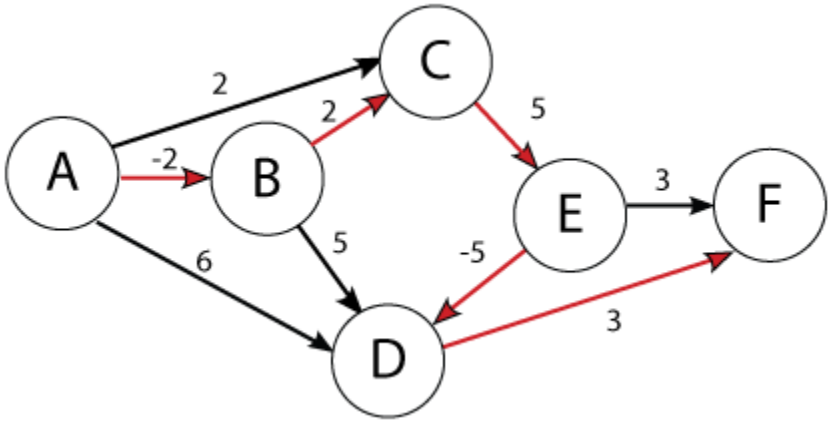
Solutions:

Shortest path: A->B->C->E-> D->F at cost 3

	A	B	C	D	E	F
0	0	INF	INF	INF	INF	INF
1	0	-2	2	6	INF	INF
2	0	-2	0	3	7	9
3	0	-2	0	2	5	6

4	0	-2	0	0	5	5
5	0	-2	0	0	5	3
6	0	-2	0	0	5	3

Many students calculated from F to A which is fine but the numbers in the table will be different even though the final path will be the same.



Question

Given the weights and values of the four items in the table below, select a subset of items with the maximum combined value that will fit in a knapsack with a weight limit, *W,* of 6. Use dynamic programming. *Show your work.*

Item $_i$	Value v_i	Weight w_i
1	3	4
2	2	3
3	4	2
4	4	3

Capacity of knapsack W=6

Solution:

Capacity of knapsack W=9
Algorithm: given two arrays w[4, 3, 2, 3] and v[3, 2, 4, 4]:

```
for I ← 1 to n:
        for x ← 1 to w:
                if w[I] > x:
                        OPT[I, x] ← OPT[i – 1, x]
                else
                        OPT[I, x] ← max(OPT[i – 1, x]; OPT[i – 1, x – w[i] + v[i])
```

6	3	3	7	8 <--
5	3	3	6	8
4	3	3	4	4
3	0	2	4 <--	4
2	0	0	4	4
1	0 <--	0 <--	0	0
	1	**2**	**3**	**4**

We used items 3, 4 for a combined value of 8 in the knapsack.

S={3,4}

Network Flow (maximum flow theory, maximum flow applications, assignment problem)

In graph theory, a flow network (also known as a transportation network) is a directed graph where each edge has a capacity and each edge receives a flow. The amount of flow on an edge cannot exceed the capacity of the edge. Often in operations research, a directed graph is called a network, the vertices are called nodes and the edges are called arcs. A flow must satisfy the restriction that the amount of flow into a node equals the amount of flow out of it, unless it is a source, which has only outgoing flow, or sink, which has only incoming flow. A network can be used to model traffic in a road system, circulation with demands, fluids in pipes, currents in an electrical circuit, or anything similar in which something travels through a network of nodes.

Ford–Fulkerson

The Ford–Fulkerson method or Ford–Fulkerson algorithm (FFA) is a greedy algorithm that computes the maximum flow in a flow network. It is called a "method" instead of an "algorithm" as the approach to finding augmenting paths in a residual graph is not fully specified or it is specified in several implementations with different running times.[2] It was published in 1956 by L. R. Ford, Jr. and D. R. Fulkerson.[3] The name "Ford–Fulkerson" is often also used for the Edmonds–Karp algorithm, which is a specialization of Ford–Fulkerson.

The idea behind the algorithm is as follows: as long as there is a path from the source (start node) to the sink (end node), with available capacity on all edges in the path, we send flow along one of the paths. Then we find another path, and so on. A path with available capacity is called an augmenting path.

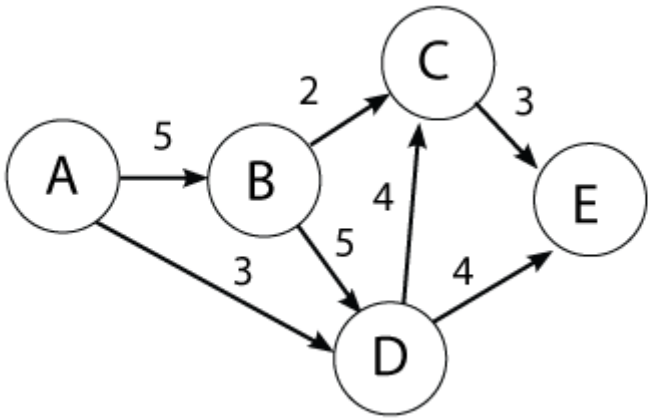
Algorithm Ford–Fulkerson

Inputs Given a Network with flow capacity , a source node , and a sink node

Output Compute a flow from to of maximum value

1. for all edges
2. While there is a path from to in , such that for all edges :
    1. Find
    2. For each edge
        1. (*Send flow along the path*)
        2. (*The flow might be "returned" later*)

Question



Use the Ford-Fulkerson algorithm to find the maximum flow from node A to E in the weighted directed graph above. *Show your work.*

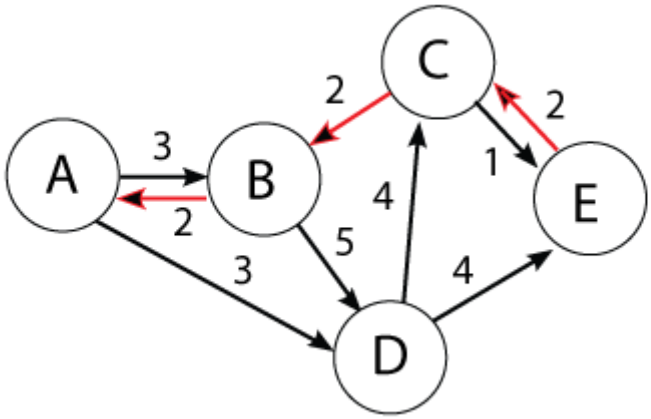
Solutions:

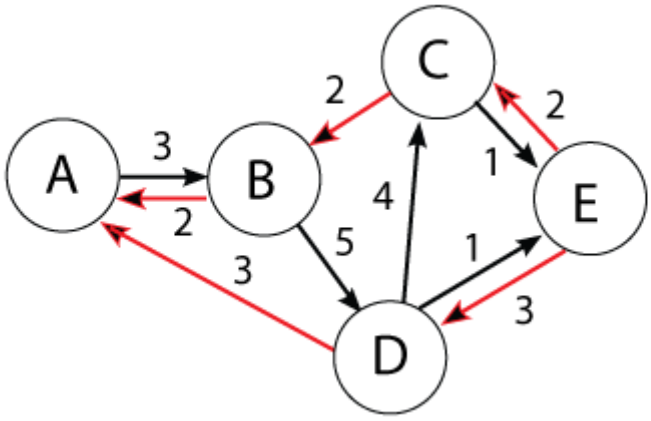
AP stands for Augmenting Path

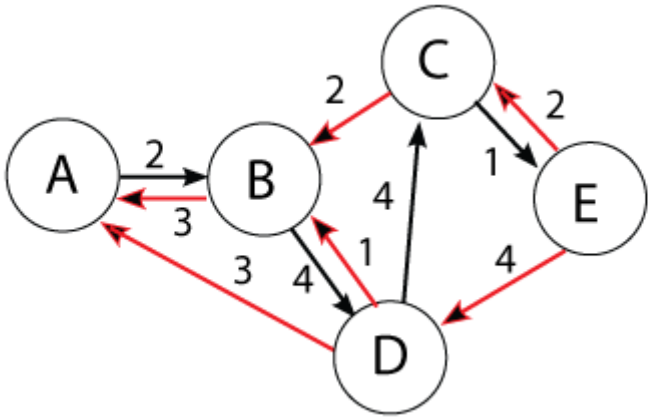
AP1: (A-B: 3, B-A: 2) -> (B-C: 0, C-B: 2) -> (C-E: 1, E-C: 2) Flow 2

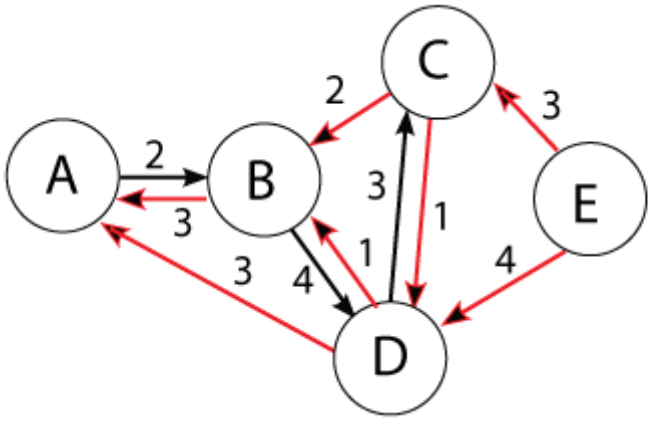
AP2: (A-D: 0, D-A: 3) -> (D-E: 1, E-D: 3) Flow 3
AP3: (A-B: 2, B-A: 3) -> (B-D: 4, D-B: 1) -> (D-E: 0, E-D: 4) Flow 1
AP4: (A-B: 1, B-A: 4) -> (B-D: 3, D-B: 2) -> (D-C: 3, C-D: 1) -> (C-E: 0, E-C: 3) Flow 1

No more augmenting paths. Max-Flow 7









Note there are many paths that could generate a Max-Flow of 7 but we can see from the cut-set of 2 edges coming in to E that the Max-Flow can't be greater than 7.

Intractability (polynomial-time reductions, P, NP, and NP-complete)

In computational complexity theory, an NP-complete decision problem is one belonging to both the NP and the NP-hard complexity classes. In this context, NP stands for "nondeterministic polynomial time". The set of NP-complete problems is often denoted by NP-C or NPC.

Although any given solution to an NP-complete problem can be verified quickly (in polynomial time), there is no known efficient way to locate a solution in the first place; indeed, the most notable characteristic of NP-complete problems is that no fast solution to them is known. That is, the time required to solve the problem using any currently known algorithm increases very quickly as the size of the problem grows. As a consequence, determining whether it is possible to solve these problems quickly, called the P versus NP problem, is one of the principal unsolved problems in computer science today.

While a method for computing the solutions to NP-complete problems using a reasonable amount of time remains undiscovered, computer scientists and programmers still frequently encounter NP-complete problems. NP-complete problems are often addressed by using heuristic methods and approximation algorithms.

Question

A contracting company sends teams to perform jobs at various sites in the Boston area. Each job takes one day to perform. The company has many teams and can easily arrange to hire more teams, but there are constraints that prevent some jobs from being performed on the same day. For example, if two jobs are at the same site, they must be performed on different days. The company wants an algorithm that can determine whether a particular set of jobs can be completed in 3 days. What complexity class does this problem have? Give a proof that it has the complexity class.

Solution:

Poly-time certifier
Given a set of jobs
Keep a two variables min start date and max end date
Loop thru n jobs
  If job start date < min start date: min start date= job start date
  If job end date > max end date: max end date = job end date
  Loop thru n-1 jobs
    Check job for conflicts with other n-1 jobs
    if any job has conflicts with another return no
If finish without any job conflicts and has max end date- min start date < 3 then return yes

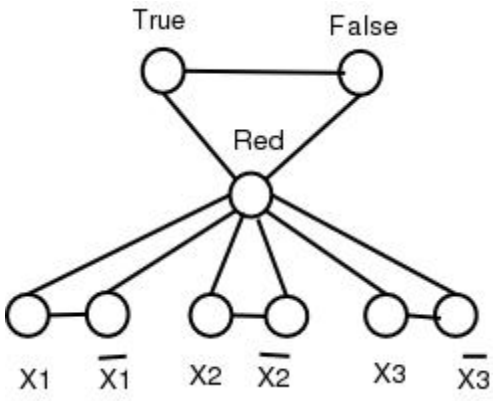
Using a graph and mapping to 3-Color you can use the 3-Color problem as your poly-time certifier.

3-Color – Given an undirected graph G does there exist a way to color the nodes red, green blue so that no adjacent nodes have the same color.
Given a graph G=(V,E). Each node is a job with a coloring, an edge represents a connection between teams. Two vertices are adjacent if there is conflict between the jobs. Given a 3-coloring, the jobs are done by performing all jobs of one color on the first day, then the ones of another color the next day, and the ones of the third color the last day. Using this graph one can get a yes/no decision to a specific solution to a 3-color graph by looping thru the vertices and checking if any adjacent nodes have the same color.

Recipe to establish NP-completeness of problem Y.
Step 1. Show that Y is in NP.
Step 2. Choose an NP-complete problem X.
Step 3. Prove that X ≤p Y.

Poly-time reduction
3-Color and Independent Set work best for this as they inherently have the notion of conflict. If you use something like 3-SAT you need to be specific about how the literals and clauses are set up.
This is equivalent to the 3-color problem which is NP-complete. A student must give a proof that the problem is equivalent to the 3-color problem (or to some other NP-complete problem).
a. Each job corresponds to a vertex.
b. Two vertices are adjacent if there is conflict between the jobs.
c. Given a 3-coloring, the jobs are done by performing all jobs of one color on the first day, then the ones of another color the next day, and the ones of the third color the last day.
d. Conversely, if one can complete the jobs in 3 days, then one can 3-color the graph.
It is only required to have a notion of conflict. Introducing a notion of "site" is neither necessary nor sufficient because site conflicts are only one example of a conflict.

Note: If you use something like 3-SAT you still need to show that it is "equivalent" to the 3-colorability problem with a construction like the one below or making a direct reduction to the problem asked.



Question

WebFlix maintains customer data in a 2D-array called WF. Where the rows correspond to the customers and the columns correspond to films that it rents. An entry WF[i,j] indicates the number of times a customer has rented a film.

WebFlix wants to find subsets of customers who have never rented the same film. (i.e. they share no entries WF[i,j] ≥ 1). WebFlix calls these Distinct Customer Subsets. We define the Distinct Customer Subset problem as follows: Given a *c* by *f* (customers by films) array of customers and films and a number k ≤ c, is there a subset of at least k customers that is *distinct?*

A. Is the Distinct Customer Subset problem NP? Why or why not?
B. Is the Distinct Customer Subset problem NP-complete? If NP-complete show a polynomial-time reduction.

Solution:

Part A

Is the Distinct Customer Subset problem NP? Why or why not?

Solution:

Yes. Because it can be verified by a nested loop. For every film count the number of customers who rented if the number is greater than 1 return “not a solution”. After the loop is complete the solution has been verified. The nested loop is polynomial because it runs in O(C F).

Part B

Is the Distinct Customer Subset problem NP-complete? If NP-complete show a polynomial-time reduction.

Solution:

Independent set can be reduced to this problem. Where every node represents a different customer and every edge represents a different film. Any node incident to an edge means that that customer has seen that film. That implies that any 2 nodes connected by the same edge have seen the same film and therefore they can't be in the same subset. A solution to this problem for at least k customers would solve the independent set problem for at least k nodes.

Approximation Algorithms (approximation algorithms)

In computer science and operations research, approximation algorithms are efficient algorithms that find approximate solutions to NP-hard optimization problems with provable guarantees on the distance of the returned solution to the optimal one. Approximation algorithms naturally arise in the field of theoretical computer science as a consequence of the widely believed P ≠ NP conjecture. Under this conjecture, a wide class of optimization problems cannot be solved exactly in polynomial time. The field of approximation algorithms, therefore, tries to understand how closely it is possible to approximate optimal solutions to such problems in polynomial time. In an overwhelming majority of the cases, the guarantee of such algorithms is a multiplicative one expressed as an approximation ratio or approximation factor i.e., the optimal solution is always guaranteed to be within a (predetermined) multiplicative factor of the returned solution. However, there are also many approximation algorithms that provide an additive guarantee on the quality of the returned solution.

Local Search (Metropolis, Hopfield nets)

In constraint satisfaction, local search is an incomplete method for finding a solution to a problem. It is based on iteratively improving an assignment of the variables until all constraints are satisfied. In particular, local search algorithms typically modify the value of a variable in an assignment at each step. The new assignment is close to the previous one in the space of assignment, hence the name local search.

All local search algorithms use a function that evaluates the quality of assignment, for example the number of constraints violated by the assignment. This amount is called the cost of the assignment. The aim of local search is that of finding an assignment of minimal cost, which is a solution if any exists.

Two classes of local search algorithms exist. The first one is that of greedy or non-randomized algorithms. These algorithms proceed by changing the current assignment by always trying to decrease (or at least, non-increase) its cost. The main problem of these algorithms is the possible presence of plateaus, which are regions of the space of assignments where no local move decreases cost. The second class of local search algorithm have been invented to solve this problem. They escape these plateaus by doing random moves, and are called randomized local search algorithms.

Question

How does the Metropolis algorithm different from Gradient descent algorithms?

Solution:

Gradient descent can occasionally make "uphill" steps.

Gradient descent
Let S denote current solution. If there is a neighbor S' of S with strictly lower cost, replace S with the neighbor whose cost is as small as possible. Otherwise, terminate the algorithm.

Metropolis algorithm
Globally biased toward "downhill" steps, but occasionally makes "uphill" steps to break out of local minima.

Metropolis algorithm (symmetric proposal distribution)

Let *f(x)* be a function that is proportional to the desired probability distribution *P(x)*.

1. Initialization: Choose an arbitrary point x_0 to be the first sample, and choose an arbitrary probability density $Q(x|y)$ which suggests a candidate for the next sample value *x*, given the previous sample value *y*. For the Metropolis algorithm, *Q*must be symmetric; in other words, it must satisfy $Q(x|y) = Q(y|x)$. A usual choice is to let $Q(x|y)$ be a Gaussian distribution centered at y, so that points closer to y are more likely to be visited next—making the sequence of samples into a random walk. The function Q is referred to as the proposal *density* or *jumping distribution*.
2. For each iteration *t*:
   - Generate a candidate *x'* for the next sample by picking from the distribution $Q(x'|x_t)$.
   - Calculate the *acceptance ratio* $\alpha = f(x')/f(x_t)$, which will be used to decide whether to accept or reject the candidate. Because *f* is proportional to the density of *P*, we have that $\alpha = f(x')/f(x_t) = P(x')/P(x_t)$.
   - If $\alpha \geq 1$, then the candidate is more likely than x_t; automatically accept the candidate by setting $x_{t+1} = x'$. Otherwise, accept the candidate with probability α; if the candidate is rejected, set $x_{t+1} = x_t$, instead.

Gradient descent
Let S denote current solution. If there is a neighbor S' of S with strictly lower cost, replace S with the neighbor whose cost is as small as possible. Otherwise, terminate the algorithm.

Gradient descent can't occasionally make "uphill" steps. Gradient descent is based on the observation that if the multivariable function $F(\mathbf{x})$ is defined and differentiable in a neighborhood of a point $\mathbf{a}$, then $F(\mathbf{x})$ decreases *fastest* if one goes from $\mathbf{a}$ in the direction of the negative gradient of F at $\mathbf{a}$, $-\nabla F(\mathbf{a})$. It follows that, if $\mathbf{b} = \mathbf{a} - \gamma \nabla F(\mathbf{a})$ for γ small enough, then $F(\mathbf{a}) \geq F(\mathbf{b})$

Randomized Algorithms (randomized algorithms)

A randomized algorithm is an algorithm that employs a degree of randomness as part of its logic. The algorithm typically uses uniformly random bits as an auxiliary input to guide its behavior, in the hope of achieving good performance in the "average case" over all possible choices of random bits. Formally, the algorithm's performance will be a random variable determined by the random bits; thus either the running time, or the output (or both) are random variables.

One has to distinguish between algorithms that use the random input so that they always terminate with the correct answer, but where the expected running time is finite (Las Vegas algorithms, example of which is Quicksort), and algorithms which have a chance of producing an incorrect result (Monte Carlo algorithms, example of which is Monte Carlo algorithm for MFAS) or fail to produce a result either by signaling a failure or failing to terminate.

Question

What is the probability of getting exactly 2 heads after flipping three coins?

Solution:

If one is head and 0 tails:
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1

There are three events in getting exactly 2 heads
{0 1 1, 1 0 1, 1 1 0}
So 3/8 is the probability of getting exactly 2 heads after flipping three coins.

We can also the formula for a discrete random variable based on a binomial distribution:

$$\binom{n}{k} p^k (1-p)^{n-k}$$

X = 1, with probability $\binom{3}{2}(\frac{1}{2})^1(\frac{1}{2})^2 = \frac{3}{8}$

Note: 3 choose 2 is 3. The 3/8 comes from 3* $1/2^3$

Question

Consider a six-sided die that gets a 1 with probability p = 1/6. How confident are you that you can get a 1 after rolling the die 3 times?

Solution:

Because these are independent events (the roll of one die doesn't affect another) we have p = 1/6 chance on each of the die throws. You want probability of at least a 1 in 3 rolls that means total probability (1 - none of the dice has 1). Hence the probability comes out to be $1-(5/6)^3=1-0.58=0.42$

However if we want exactly one success (a roll of 1) in three tries this is just the binomial expansion. If we run n trials, where the probability of success for each single trial is p, what is the probability of exactly k successes?

$$\frac{1-p}{1} \quad \frac{p}{2} \quad \frac{p}{3} \quad \frac{1-p}{4} \quad \frac{p}{5} \quad \ldots\ldots \quad \frac{}{n}$$

k slots where prob. success is p , n-k slots where prob. failure is $1-p$

Thus, the probability of obtaining a specific configuration as denoted above is $p^k(1-p)^{n-k}$. From here, we must ask ourselves, how many configurations lead to exactly k successes. The answer to this question is simply, "the number of ways to choose k slots out of the n slots above. This is $\binom{n}{k}$. Thus, we must add $p^k(1-p)^{n-k}$ with itself exactly $\binom{n}{k}$ times.

This leads to the formula:

$$\binom{n}{k} p^k (1-p)^{n-k}$$

X = 1, with probability $\binom{3}{1}(\frac{1}{6})^1(\frac{5}{6})^2 = \frac{25}{72}$

Note: 3 choose 1 is 3

25/72 = 0.3472 see WolframAlpha - (3 choose 1)* 1/6*(5/6)^2 http://po.st/ZztZy1

Question

Hat-check problem. Each of n customers gives a hat to a hat-check person at a restaurant. The hat-check person gives the hats back to the customers in a random order. What is the expected number of customers who get back their own hat?

Solution:

Probability of getting back your hat (1/n) if n customers. The expected number of customers that get back their own hat is (number of customers)*P(that customer get back his own hat. That is, n * (1/n) = 1. In other words 1 customer is expected to get back their hat no matter the number of customers.

Question

Consider a server cluster with up to 2n jobs but only two servers. The load balancer uniformly (at random) gives a new job to one of the two servers.

A. (2 points) What is the expected number of jobs for each server?
B. (4 points) Let X_1 and X_2 denote random variables representing the number of jobs in servers 1 and 2. How big can the difference $(X_1 – X_2)$ grow?
C. (4 points) Can we bound the difference in growth?

Solution:

A. (2 points) What is the expected number of jobs for each server?

2n for two servers, *n* per server.

B. (4 points) Let X_1 and X_2 denote random variables representing the number of jobs in servers 1 and 2. How big can the difference $(X_1 – X_2)$ grow?

Note: I gave 3 of 4 points to say the difference was 2n even though this isn't right.

The point of this question was to show that the difference $(X_1 – X_2)$ decreases exponentially as n increases. 2n is not an exponentially decreasing function. If you said 2n and want to get back that 1 point you can write a program that for LARGE n actually reaches 2n and show it to me before next Wednesday.

To show that the difference $(X_1 – X_2)$ decreases exponentially as n increases you can choose a δ of +/- 1 times the mean, and plug that into the upper/lower Chernoff bounds. If $\mu = n$ and $X_1 – X_2$ can +/- n, then this gives us a $\delta = 1$ and $\mu = n$. ($\delta = n$ and $\mu = n$ was also accepted for full credit)

The Chernoff bound (above mean):

$Pr[X>(1+\delta)\mu] < [e^{\delta}/(1+\delta)^{(1+\delta)}]^{\mu}$

For $\delta = 1$ and $\mu = n$.

$Pr[X>(1+1)n] < [e^{1}/(1+1)^{(1+1)}]^{n}$

The Chernoff bound (below mean)

$Pr[X<(1-\delta)\mu] < [e^{-\delta^2\mu/2}/(1+\delta)^{(1+\delta)}]^{\mu}$

For $\delta = 1$ and $\mu = n$.

$Pr[X<(1-1)n] < [e^{-1^2 n/2}/(1+1)^{(1+1)}]^{n}$

Also a $\delta = n$ and $\mu = n$ was also accepted for full credit.

$Pr[X>(1+n)n] < [e^{n}/(1+n)^{(1+n)}]^{n}$

$Pr[X<(1-n)n] < [e^{-n^2 n/2}/(1+n)^{(1+n)}]^{n}$

C. (4 points) Can we bound the difference in growth?

Yes. You can choose a δ, say 0.05 or 0.1. Say we pick 0.1 and we know the mean is n.

The Chernoff bound (above mean):

$Pr[X>(1+\delta)\mu] < [e^{\delta}/(1+\delta)^{(1+\delta)}]^{\mu}$

For $\delta = 0.1$ and $\mu = n$.

$Pr[X>(1+0.1)n] < [e^{0.1}/(1+0.1)^{(1+0.1)}]^{n}$

The Chernoff bound (below mean)

$Pr[X<(1-\delta)\mu] < [e^{-\delta^2\mu/2}/(1+\delta)^{(1+\delta)}]^{\mu}$

For $\delta = 0.1$ and $\mu = n$.

$Pr[X<(1-0.1)n] < [e^{-0.1^2 n/2}/(1+0.1)^{(1+0.1)}]^{n}$