Prep for Comprehensive Exam in Algorithms

Brad Burkman

18 August 2019

Contents

Index

0.1 Topics Remaining

- Fibonacci Heaps (S18 #L2)
- What is the preferred data structure of implementing binary heap, also justify your answer. (S17 #S5)
- Spring 2015 #L1.d
  Consider the following pseudo code for Kruskal's algorithm for solving minimal spanning tree (MST).
  Algorithm MST. Let N be the number of nodes in graph G.

```
1 Sort the edges in non-decreasing order of cost.
2 T is an empty graph
3 while T has fewer than N-1 edges do:
4         let e denote the next edge of G (in the order of cost)
5         if T∪{e} does not contain a cycle, then T=T∪{e}
```

  Clearly mentioning the data structure you have to employ to reduce the time complexity to access and to maintain the necessary information, show the exact time taken to obtain the MST. Also show the tight bound of the algorithm. (Pay attention in detecting a cycle.)

0.2 Take to Exam

- Ruler
- Calculators
- Water
- Good eraser
-

0.3 Definitions

Articulation Node (or Articulation Point), Problem 22-2, page 621). Let $G = (V, E)$ be connected, undirected graph. An *articulation point* of G is a vertex whose removal disconnects G. A *bridge* of G is an edge whose removal disconnects G.

Chapter 1

Time Complexity

1.1 Time Complexity

1.1.1 Definitions

Big-Ω Asymptotic lower bound

$\Omega(g(n)) = \{f(n) :$ there exist positive constants c and n_0 such that $0 \leq c \cdot g(n) \leq f(n)$ for all $n \geq n_0\}$

Big-O Asymptotic upper bound

$O(g(n)) = \{f(n) :$ there exist positive constants c and n_0 such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0\}$

Big-Θ Asymptotic tight bound

$\Theta(g(n)) = \{f(n) :$ there exist positive constants c_1, c_2, and n_0 such that $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ for all $n \geq n_0\}$

For any two functions $f(n)$ and $g(n)$, we have

$$f(n) = \Theta(g(n)) \iff f(n) = \Omega(g(n)) \text{ and } f(n) = O(g(n))$$

1.1.2 Algorithms

Sorting Algorithms

$\Omega(n \lg n)$	Any comparison sort.
$\Theta(n^2)$	Insertion Sort
$\Theta(n \lg n)$	Merge Sort
$O(n \lg n)$	Heap Sort
$\Theta(n^2)$	Quicksort worst time
$\Theta(n \lg n)$	Quicksort average time, assuming distinct elements.

Graph Algorithms

$O(V+E)$	BFS
$\Theta(V+E)$	DFS
$\Theta(V+E)$	Strongly Connected Components, because it's DFS
$O(E \lg V)$	Minimum Spanning Tree with either Kruskal or Prim
$O(V^2)$	Dijkstra's Algorithm with vertices in an array
$O(E \lg V)$	Dijkstra with vertices in a binary min-heap
$O(E\|f*\|)$	Ford-Fulkerson if $f*$ is a maximum flow.

1.1.3 Old Exam Problems

Spring 2019 #S1

Give a big-O (upper bound) estimate for $f(n) = n\log(n!) + 3n^2 + 2n - 10000$, where n is a positive integer.

Solution

$$\begin{aligned} n! &= n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 2 \cdot 1 \\ \log(n!) &= \log(n) + \log(n-1) + \log(n-2) + \cdots + \log(2) + \log(1) \\ &\leq \log(n) + \log(n) + \log(n) + \cdots + \log(n) + \log(n) \\ &= n\log(n) \\ n\log(n!) &\leq n^2 \log(n) \end{aligned}$$

Claim: $f(n)$ is $O(n^2 \log n)$.
To prove the claim, we need to show that there exist positive constants c and n_0 such that

$$0 \leq n^2 \log n + 3n^2 + 2n - 10000 \leq c \cdot n^2 \log n \text{ for all } n \geq n_0$$

Choose $c = 6$ and $n_0 = 100$. For $n \geq n_0$, $f(n) > 0$, and

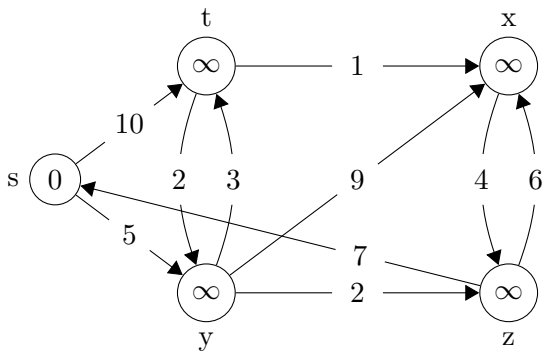
$$
\begin{aligned}
n &\leq n^2 \log n \\
2n &\leq 2n^2 \log n \\
n^2 &\leq n^2 \log n \\
3n^2 &\leq 3n^2 \log n \\
n^2 \log n + 3n^2 + 2n - 10000 &\leq n^2 \log n + 3n^2 \log n + 2n^2 \log n - 10000 \\
&= 6n^2 \log n - 10000 \\
&\leq 6n^2 \log n
\end{aligned}
$$

Since there exist constants c and n_0 such that $f(n) \leq c \cdot n^2 \log n$ for all $n \geq n_0$, $f(n) = O(n^2 \log n)$.

Spring 2019 #L4

The Dijkstra's algorithm (DIJ) solves the single-source shortest-path problem in a weighted directed graph $G = (V, E)$. Given the graph G below, follow DIJ to find shortest paths from vertex s to all other vertices, with all predecessor edges shaded and estimated distance values from s to all vertices provided at the end of each iteration.

What is the time complexity of DIJ for a general graph $G = (V, E)$, if the candidate vertices are kept in a binary min-heap?



Solution to Time Complexity Part

The time complexity for Dijkstra's algorithm with the vertices stored in an array is $O(V^2)$. If the vertices are stored in a binary min-heap, the time complexity is $O(E \lg V)$, which is faster if the graph is sparse. If the graph is complete, $E = V(V-1)/2$, which would make the binary min-heap option $O(V^2 \lg V)$, worse than the array option.

Fall 2018 #S1

The divide and conquer strategy (D&C) has been used to solve problem efficiently to reduce the overall computational cost to certain types of problems.

a. Which conditions have to be satisfied for D&C to solve such problems successfully? (Clearly state.)
b. Suppose the size of a problem involved in D&C is $n = 2^k$. Let the cost in dividing the problems into an equal size is constant and the time to combine solutions to sub-problems is linear. Write the recurrence relations and then find the tight bound in solving such problems using D&C.

Solution

Divide and conquer should be used (instead of dynamic programming) when the subproblems will not be solved only once.

a. The steps for divide and conquer are:

   Divide the problem into subproblems.
   Conquer by recursively solving the subproblems.
   Combine the solutions.

b. Let $T(n)$ be the running time of the algorithm for a problem of size n.
   Let a be the (small) size of the problem for which the solution time is constant.
   Let d be the constant cost of dividing the problems.
   Let cn be the linear cost of combining the solutions to the subproblems.

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le a \\ 2T\left(\frac{n}{2}\right) + d + cn & \text{otherwise} \end{cases}$$

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + d + cn \\ &= 2\left[2T\left(\frac{n}{4}\right) + d + c\frac{n}{2}\right] + d + cn = 4T\left(\frac{n}{4}\right) + 3d + 2cn \\ &= 4\left[2T\left(\frac{n}{8}\right) + d + c\frac{n}{4}\right] + 3d + 2cn = 8T\left(\frac{n}{8}\right) + 7d + 3cn \\ &= \cdots = 2^k T\left(\frac{n}{2^k}\right) + (2^k - 1)d + kcn \\ &\text{Terminates when } n = 2^k \iff k = \lg n. \\ &= (n-1)d + cn\lg n \\ &= \Theta(n \lg n) \end{aligned}$$

Fall 2018 $S2

a. What is the lower bound for comparisons based sorting algorithm? (Outline the justification of your answer.)
b. What is the strategy behind greedy algorithm?

Solution

a. The lower bound (in the worst case) to sort n elements by any comparison sort is $\Omega(n \lg n)$, because it has to make that many comparisons.
b. The strategy behind a greedy algorithm is to find optimal solutions to subproblems, and build those together into an optimal solution to the original problem. In order to work the problem must have the greedy choice property, that choosing locally optimal solutions leads to a globally optimal solution.

Fall 2018 #S3

Find the tight bounds (by deriving their upper and lower bounds) of the following expressions.

a. $T(n) = 2 \cdot T\left(\frac{n}{8}\right) + n^{\frac{1}{3}}$
b. $T(n) = \log(n!)$

Solutions

a.

$$
\begin{aligned}
T(n) &= 2T\left(\frac{n}{8}\right) + n^{1/3} \\
T\left(\frac{n}{8}\right) &= 2T\left(\frac{n}{8^2}\right) + \left(\frac{n}{8}\right)^{1/3} \\
T\left(\frac{n}{8^2}\right) &= 2T\left(\frac{n}{8^3}\right) + \left(\frac{n}{8^2}\right)^{1/3} \\
&\vdots \\
T(n) &= 2T\left(\frac{n}{8}\right) + n^{1/3} \\
&= 2\left(2T\left(\frac{n}{8^2}\right) + \left(\frac{n}{8}\right)^{1/3}\right) + n^{1/3} = 4T\left(\frac{n}{8^2}\right) + 2n^{1/3} \\
&= 4\left(2T\left(\frac{n}{8^3}\right) + \left(\frac{n}{8^2}\right)^{1/3}\right) + 2n^{1/3} = 8T\left(\frac{n}{8^3}\right) + 3n^{1/3} \\
&= \log_8 n \cdot n^{1/3} \\
T(n) &= \Theta(\sqrt[3]{n} \cdot \lg n)
\end{aligned}
$$

Because we didn't use any inequalities, we get Big-Θ.

b. Upper bound:

$$\begin{aligned}\log(n!) &= \log(1 \cdot 2 \cdot \cdots \cdot n) \\ &= \log 1 + \log 2 + \cdots + \log n \\ &\leq \log n + \log n + \cdots + \log n \\ &= n \log n\end{aligned}$$

So $\log(n!) = O(n \log n)$.
Lower bound:

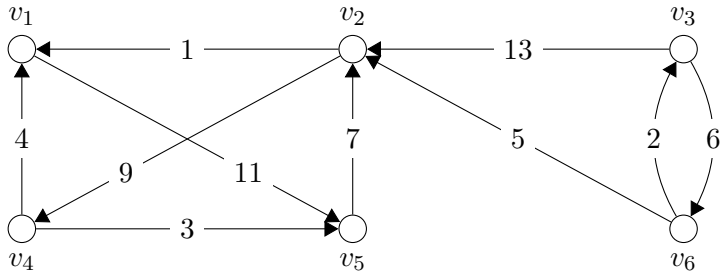
$$\begin{aligned}\log(n!) &= \log(1 \cdot 2 \cdot \cdots \cdot \frac{n}{2} \cdot \cdots \cdot n) \\ &= \log 1 + \log 2 + \cdots + \log \frac{n}{2} + \cdots + \log n \\ &\geq \log \frac{n}{2} + \log \left(\frac{n}{2} + 1\right) + \cdots + \log n \\ &\geq \log \frac{n}{2} + \log \frac{n}{2} + \cdots + \log \frac{n}{2} \\ &= \frac{n}{2} \log \frac{n}{2} \\ &= \frac{n}{2}(\log n - \log 2) \\ &\geq \frac{1}{4} n \log n \qquad \text{for } n \geq 4\end{aligned}$$

So $\log(n!) = \Omega(n \log n)$.
Since $\log(n!) = \Omega(n \log n)$ and $\log(n!) = O(n \log n)$, therefore $\log(n!) = \Theta(n \log n)$.

Fall 2018 #L4

The Dijkstra's algorithm (DS) solves the single-source shortest-path problem in a weighted graph $G = (V, E)$ without negative weighted edges or cycles, by edge relaxation at one vertex at a time until all vertices are examined. Given the graph G below, follow DS to find the shortest paths from vertex v_1 to all other vertices, with all predecessor edges shaded and estimated distance values from v_1 to all vertices provided at the end. Also list the sequence of vertices at which relaxation takes place.

What is the time complexity of DS for a general graph $G = (V, E)$ when candidate vertices are kept in an array?



Solution to Time Complexity Part

The time complexity of Dijkstra's algorithm, if the candidate vertices are kept in an array, is $O(V^2)$. If the candidate vertices are kept in a binary min-heap, then the time complexity is $O(E \lg V)$, which is worse if the graph is dense or complete, but better if the graph is sparse.

Spring 2018 #S1

A problem with size n follows a typical divide-and-conquer approach to have its time complexity of

$$T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + c \cdot n$$

Solve $T(n)$. (Show your work.)

Solution

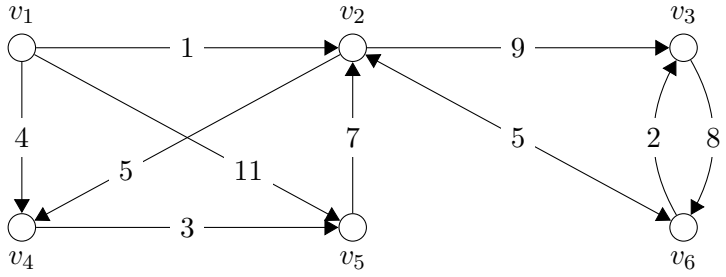
$$\begin{aligned}
T(n) &= T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + c \cdot n \\
&= \left[T\left(\frac{n}{16}\right) + T\left(\frac{3n}{16}\right) + \frac{cn}{4}\right] + \left[T\left(\frac{3n}{16}\right) + T\left(\frac{9n}{16}\right) + \frac{3cn}{4}\right] + cn \\
&= T\left(\frac{n}{4^2}\right) + 2T\left(\frac{3n}{4^2}\right) + T\left(\frac{3^2}{4^2}n\right) + 2cn \\
&\quad \text{The recursion terminates when } \left(\frac{3}{4}\right)^k n = 1 \text{ in step } k = \log_{4/3} n \\
T(n) &= (\log_{4/3} n)(c)(n) \\
T(n) &= O(n \log n)
\end{aligned}$$

Spring 2018 #S4

The Dijkstra's algorithm (DS) solves the single-source shortest-path problem in a weighted graph $G = (V, E)$ without negative weighted edges or cycles, by edge relaxation at one vertex at a time until all vertices are examined. Given the graph G below, follow DS to find the shortest paths from vertex v_1 to all other vertices, with all predecessor edges shaded and estimated distance values from

v_1 to all vertices provided at the end. Also list the sequence of vertices at which relaxation takes place.

What is the time complexity of DS for a general graph $G = (V, E)$ when candidate vertices are kept in an array?



Solution to the Time Complexity Part

When the vertices are kept in an array, the time complexity of Dijkstra's algorithm is $O(V^2)$. When the vertices are kept in a binary min-heap, the time complexity is $O(E \lg V)$. In a complete graph, $E = V(V-1)/2$, so for a complete (or even dense) graph, the array is faster, but for a sparse graph, the binary min-heap is faster.

Spring 2018 #L3

a. To what extent the asymptotic upper bound and lower bound provide insight on running time of an algorithm.

b. Compare and contrast asymptotic tight bound to the average running time of an algorithm.

c. Consider the pseudo code of an algorithm given below.

   (a) What the value K in line 4 denote?

   (b) What the value m in line 8 denote?

   (c) When the algorithm terminates, what does the value $m + K$ in line 9 denote?

   (d) Find the asymptotic tight bound of Algorithm Test below.

```
    AlgorithmTest(n)
1   K = 0
2   for i = 1 to n
3           for j = 1 to i
4                   K = K + 1
5   m = 0
6   for i = 1 to n - 1
7           for j = i + 1 to n
8                   m = m + 1
9   return (m + K)
```

Solution

a. The asymptotic lower bound gives the order of magnitude of the best-case running time of an algorithm for sufficiently large input size; for instance, the time to run a sorting algorithm on a list that is already sorted. The asymptotic upper bound gives the worst-case run time. The bounds do not give the actual run time, because there is a (perhaps unknown) constant not given. The bounds show how the run time increases with input size. If the upper bound of algorithm A is of lower order than the lower bound of algorithm B, there is an input size for which A is always faster than B, but the input size may be larger than an actual application.

b. There are some actual applications for which we have algorithms with an asymptotic tight bound, like multiplication of dense square matrices of uniform data types (like double-precision floats). For matrices of size $n \times n$, the algorithm has asymptotic tight bound $\Theta(n^3)$, meaning that, for sufficiently large n, there exist constants c_1 and c_2 such that the algorithm takes between $c_1 n^3$ and $c_2 n^3$ units of time, but c_1 and c_2 are not given, and in practice depend on everything from the hardware to the weather. The asymptotic tight bound also makes the heroic assumption that the data needed is always in the registers, when in fact, for sufficiently large n, the data may not fit in cache or even in main memory. The asymptotic tight bound is not very useful in predicting average running time of a particular instance, but is useful in comparing algorithms and understanding how the average running time will increase with the size of the input.

c. (a) The value of K denotes the arithmetic series $1 + 2 + 3 + \cdots + n = (n)(n+1)/2$.
   (b) The value of m denotes the arithmetic series $(n-1)+(n-2)+\cdots+2+1 = (n-1)(n)/2$.
   (c) The value of $m + K$ is n^2.
   (d) The asymptotic tight bound is $\Theta(n^2)$, which is the number of additions and the number of calls to the inner loops. The number of calls to the outer loops is $O(n)$, and there's some constant overhead.

Spring 2017 #S1

a. Define height balanced binary tree.
b. Write a pseudo code to determine whether a tree is height balanced?
c. Obtain the tight bound of your algorithm.

Solutions

a. A height balanced binary tree is a binary tree such that, for each node x, the heights of the left and right subtrees of x differ by at most 1. Alternately, the depths of the leaves differ by no more than 1.

b. Run BFS from the root. First use the results to find the largest distance from the root to another node. This largest distance will be the height of the tree, h. Use the BFS results a second time to count the nodes whose depth (distance from the root) is $h-1$. Iff that number of nodes is 2^{h-1}, then the tree is height balanced. For example, if the

c. The time complexity of BFS is $O(V+E)$. In a binary tree, $E = V-1$ and $V \leq 2^{h+1}-1 < 2 \cdot 2^h$, so the time complexity of BFS on a binary tree is $O(V) = O(2^h)$. It would be nice if the time complexity for BFS on a binary tree were less than for a general tree, but to get the depths of the leaf nodes we have to visit every node, and there are V of them, so we can't hope for time complexity less than linear in V. Tracking the maximum distance and counting the number of nodes with depth $n-1$ are at worst linear in V, so the time complexity of the algorithm is $\Theta(V) = \Theta(2^h)$.

Spring 2017 #S2

Suppose you have keys of N objects stored in an array in the ascending order of key values. Also assume that there is duplicate entry in the key.

a. Describe an efficient algorithm with the pseudo code that helps you to search for the object given the key. (The algorithm must return null value if the key is not in the array.)
b. Obtain the tight upper bound for your algorithm.

Solutions

a. Take a divide-and-conquer approach, bracket-and-halving.
   Let N be the number of objects, k the key of the object we seek, and A the array.
   Assume that all division truncates.
   Let $a = 1$ and $c = N$.
   At each step,

   1. $b = (a + c)/2$
   2. if $A[b] == k$, return b
   3. if $a == c$, return NULL
   4. if $A[b] < k$, then $a = b + 1$
   5. if $k < A[b]$, then $c = b - 1$

b. The tight upper bound for the algorithm is $O(\lg N)$.

Spring 2017 #S4

Find the tight for the recurrence relation below without using the master theorem (show all the steps):

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

$$
\begin{aligned}
T(n) &= T\left(\frac{n}{2}\right) + n \\
&= \left[T\left(\frac{n}{4}\right) + \frac{n}{2}\right] + n = T\left(\frac{n}{4}\right) + \frac{3}{2}n \\
&= \left[T\left(\frac{n}{8}\right) + \frac{n}{4}\right] + \frac{3}{2}n = T\left(\frac{n}{8}\right) + \frac{7}{4}n \\
&= \cdots = T\left(\frac{n}{2^k}\right) + \frac{2^k - 1}{2^{k-1}}n \\
&\quad \frac{n}{2^k} = 1 \text{ when } k = \log_2 n \\
&= (\log_2 n)\left(\frac{2^{\log_2 n} - 1}{2^{\log_2 n - 1}}n\right) = (\log_2 n)\left(\frac{n-1}{n/2}\right)n = (\log_2 n)(2)(n-1) \\
&= \Theta(n \log n)
\end{aligned}
$$

Fall 2016 #S1

1. Define an upper and the tight time bound of an algorithm.
2. In which way the average time bound will add more value to the tight bound?

Solutions

1. The time bounds are a function of the size of the inputs into an algorithm, such that the running time should be proportional to the value of the function for the input size. Because they are proportional, we can ignore constants. The upper bound gives the worst-case scenario, and the lower bound gives the best-case. If the ratio of the upper to the lower bound does not depend on the input size, then the tight bound exists.
2. For a given input size, the running time is proportional to the tight bound, but the constant of proportionality is unknown. The average time bound may give more information, equivalent to an estimate of the constant.

Fall 2016 #S2

1. Briefly describe a quick sort algorithm for sorting objects in an ascending order of their keys.
2. What is the best and worst case time complexity of quick sort and the reason for such complexity?

Solution

1.
2. Worst case: $\Theta(n^2)$.

Fall 2016 #S3

Find the tight [time bound, time complexity?] for the recurrence relations without using the master theorem.

1. $T(n) = T(n-2) + 2\lg(n)$
2. $T(n) = T(\sqrt{n}) + \lg(n)$

Solution

1. $T(n) = T(n-2) + 2\lg n$

$$
\begin{aligned}
T(n) &= T(n-2) + 2\lg(n) \\
&= [T(n-4) + 2\lg(n-2)] + 2\lg(n) = T(n-4) + 2\lg[(n)(n-2)] \\
&= [T(n-6) + 2\lg(n-4)] + 2\lg[(n)(n-2)] = T(n-6) + 2\lg[(n)(n-2)(n-4)] \\
&= \cdots = T(0) + 2\lg[(n)(n-2)(n-4)\cdots(4)(2)] \\
T(n) &= 2\lg(n) + 2\lg(n-2) + \cdots + 2\lg 2
\end{aligned}
$$

Upper bound

$$
\begin{aligned}
T(n) &= 2\lg(n) + 2\lg(n-2) + \cdots + 2\lg 2 \\
&\leq 2\lg n + 2\lg n + \cdots + 2\lg n \\
&= 2\frac{n}{2}\lg n \\
&= n\lg n \\
T(n) &= O(n\lg n)
\end{aligned}
$$

Lower bound

$$
\begin{aligned}
T(n) &= 2\lg(n) + 2\lg(n-2) + \cdots + 2\lg(n/2) + 2\lg(n/2-2) + \cdots + 2\lg 2 \\
&\geq 2\lg(n) + 2\lg(n-2) + \cdots + 2\lg(n/2) \\
&= 2\lg n + 2\lg n + \cdots 2\lg n \\
&\geq 2\frac{n}{4}\lg n = \frac{1}{2}n\lg n \\
T(n) &= \Omega(n\lg n)
\end{aligned}
$$

Since $T(n) = O(n\lg n)$ and $T(n) = \Omega(n\lg n)$, $T(n) = \Theta(n\lg n)$.

2. $T(n) = T(\sqrt{n}) + \lg n$

$$\begin{aligned}
T(n) &= T\left(n^{\frac{1}{2}}\right) + \lg(n) \\
&= \left[T\left(n^{\frac{1}{4}}\right) + \lg\left(n^{\frac{1}{2}}\right)\right] + \lg(n) = T\left(n^{\frac{1}{4}}\right) + \lg n + \frac{1}{2}\lg n \\
&= T\left(n^{\frac{1}{2^k}}\right) + \lg n + \frac{1}{2}\lg n + \cdots + \frac{1}{2^{k-1}}\lg n \\
&= T\left(n^{\frac{1}{2^k}}\right) + \left(1 + \frac{1}{2} + \cdots + \frac{1}{2^{k-1}}\right)\lg n \\
&= T\left(n^{\frac{1}{2^k}}\right) + \left(\frac{2^k - 1}{2^{k-1}}\right)\lg n
\end{aligned}$$

This problem is different from the others in that we can't directly find how many steps we'd need to take to get the argument of T to be 1, because that only happens with rounding, but we can find where the argument is 2.
Solve for k when the argument is 2, as an approximation of how many steps it takes to get to an argument of 1.
Alternately, rather that 2 we could use any constant, which would become the base of the log in the solution, so the solution would be within that constant of the number of steps.

$$\begin{aligned}
n^{\frac{1}{2^k}} &= 2 \\
\frac{1}{2^k}\log_2 n &= 1 \\
\log_2 n &= 2^k \\
\log_2(\log_2 n) &= k
\end{aligned}$$

$$
\begin{aligned}
T(n) &= T\left(n^{\frac{1}{2^k}}\right) + \left(\frac{2^k - 1}{2^{k-1}}\right) \lg n \\
&= T(2) + \left(\frac{\log_2 n - 1}{\log_2 n)/2}\right) \lg n \\
&= T(2) + \left(\frac{\log_2 n - 1}{\log_2 n)/2}\right) \lg n \\
&= T(2) + 2(\log_2 n - 1) \\
&\text{Because there was no inequality, we get } \Theta. \\
&= \Theta(\lg n)
\end{aligned}
$$

Fall 2016 #S5

Suppose there are n clauses and m variables (propositions) in a given $3 - p$ sat problem.

- How many possible interpretations are there?
- Find the tight bound of checking for satisfiability of the n clauses.

Fall 2015 #S1

Show that for arbitrary real constants a and b with $b > 0$, we have $(n + a)^b = \Theta(n^b)$.

Solution

$f(n) = \Theta(g(n))$ iff there exist constants c_1, c_2, and n_0 such that for all $n \geq n_0$,

$$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$$

Show that there exist positive constants c_1, c_2, and n_0 such that, for all $n \geq n_0$,

$$0 \leq c_1 n^b \leq (n + a)^b \leq c_2 n^b$$

In the following calculations, assume $n \geq \max(-a, 1)$ so that both n and $n + a$ are positive. Then because both sides of the inequality are positive and the logarithm is an increasing function, taking the log of both sides preserves the inequality.

$$
\begin{aligned}
c_1 n^b &\le (n+a)^b \\
\log(c_1 n^b) &\le \log\left((n+a)^b\right) \\
\log c_1 + b\log n &\le b\log(n+a) \\
\log c_1 &\le b\log(n+a) - b\log n \\
\log c_1 &\le b\log\frac{n+a}{n} \\
c_1 &\le \left(\frac{n+a}{n}\right)^b
\end{aligned}
$$

Similarly for c_2, so $c_1 \le \left(\frac{n+a}{n}\right)^b \le c_2$.

If $a \ge 0$, then $\frac{n+a}{n}$ decreases towards 1 as $n \to \infty$ and $\min\left(\frac{n+a}{n}\right)^b = 1$.

If, however, $a < 0$, then $\frac{n+a}{n}$ increases towards 1 as $n \to \infty$, and its min value in the range $[n_0, \infty)$ is $\frac{n_0+a}{n_0}$.

I will choose $n_0 = |a| + 1$ so that $\min\left(\frac{n_0+a}{n_0}\right) = \frac{1}{|a|+1}$ and $\max\left(\frac{n_0+a}{n_0}\right) = \frac{2|a|+1}{|a|+1} < 2$

Let $n_0 = |a| + 1$, $c_1 = \left(\frac{1}{|a|+1}\right)^b$ and $c_2 = 2^b$.

Then for all $n \ge n_0$, $0 \le c_1 n^b \le (n+a)^b \le c_2 n^b$, so $(n+a)^b = \Theta(n^b)$.

Fall 2015 #S2

Use the recursion-tree technique to derive the tight lower and upper bounds of the recursion $T(n) = T(n/3) + T(2n/3) + cn$.

[Note that this exercise is the example on page 91.]

Solution

$$
\begin{aligned}
T(n) &= T\left(\frac{1}{3}n\right) + T\left(\frac{2}{3}n\right) + cn \\
&= \left[T\left(\frac{1}{9}n\right) + T\left(\frac{2}{9}n\right) + \frac{1}{3}cn\right] + \left[T\left(\frac{2}{9}n\right) + T\left(\frac{4}{9}n\right) + \frac{2}{3}cn\right] + cn \\
&= T\left(\frac{1}{9}n\right) + 2T\left(\frac{2}{9}n\right) + T\left(\frac{4}{9}n\right) + 2cn \\
&= \left[T\left(\frac{1}{27}n\right) + T\left(\frac{2}{27}n\right) + \frac{1}{9}cn\right] + 2\left[T\left(\frac{2}{27}n\right) + T\left(\frac{4}{27}n\right) + \frac{2}{9}cn\right] + \left[T\left(\frac{4}{27}n\right) + T\left(\frac{8}{27}n\right) + \frac{4}{9}cn\right] + 2c \\
&= T\left(\frac{1}{27}n\right) + 2T\left(\frac{2}{27}n\right) + 2T\left(\frac{4}{27}n\right) + T\left(\frac{8}{27}n\right) + 3cn \\
&= \cdots = T\left(\frac{1}{3^k}n\right) + 2T\left(\frac{2}{3^k}n\right) + 2T\left(\frac{2^2}{3^k}n\right) + \cdots + 2T\left(\frac{2^{k-1}}{3^k}n\right) + T\left(\frac{2^k}{3^k}n\right) + kcn
\end{aligned}
$$

Last step when $\frac{2^k}{3^k}n = 1,$ which happens when $k = \log_{3/2} n$

$$
\begin{aligned}
&= \log_{3/2} n \cdot c \cdot n \\
&= \Theta(n \lg n)
\end{aligned}
$$

Fall 2015 #L3

In the following pseudo code,

a. Write the formula to determine the number of add operations at line 5 when the algorithm terminates.
b. What is the space complexity?
c. Find the following time bounds of this algorithm: Upper, lower, and tight. (Must show all the details of your work.)

```
Algorithm Count
1  Cnt = 0
2  for i = 1 to n do {
3          for j = 1 to i^2 do {
4                  for k = 1 to j do {
5                          Cnt = Cnt + 1
6                  }
7          }
8  }
```

Note that I have corrected (?) line 3 from `for j=i^2 to i do {`
This question is weird.

Solution?

$$\begin{aligned} Cnt &= \sum_{i=1}^{n} \sum_{j=1}^{i^2} j \\ &= 1 + (1+2+3+4) + (1+2+\cdots+9) + \cdots + \frac{n^2(n^2+1)}{2} \\ &\leq n \cdot \frac{n^2(n^2+1)}{2} \\ &\leq n^5 \end{aligned}$$

The number of add operations is the same as the value of Cut. I've found a upper bound.

Spring 2015 #S1

Briefly define upper, lower, and tight time bound of an algorithm. How does an average time complexity related to any of these bounds?

Solution

See Fall 2016 #S1.

Spring 2015 #S2

In terms of run time efficiency, compare and contrast quick sort and merge sort. What is the best and the worst case time complexity of the quick sort algorithm? Also state under what conditions one may expect these two extreme cases.

Spring 2015 #S3

Find the tight bounds of the following sums.

1. $\sum_{i=1}^{n} i^3 a^i$ where a is a constant greater than 1
2. $\sum_{i=1}^{n} \log(i^3)$

Solution

1. No idea.

2. Upper bound:

$$\begin{aligned}
\sum_{i=1}^{n} \log(i^3) &= 3\sum_{i=1}^{n} \log(i) \\
&= 3(\log 1 + \log 2 + \cdots + \log n) \\
&= 3\log(1 \cdot 2 \cdot \cdots \cdot n) \\
&= 3\log(n!) \\
&\leq 3\log(n^n) \\
&= 3n \log n \\
\sum_{i=1}^{n} \log(i^3) &= O(n \log n)
\end{aligned}$$

Lower bound:

$$\begin{aligned}
\sum_{i=1}^{n} \log(i^3) &= 3\sum_{i=1}^{n} \log(i) \\
&= 3(\log 1 + \log 2 + \cdots + \log(n/2) + \cdots + \log n) \\
&\geq 3(\log(n/2) + \log(n/2+1) + \cdots + \log n) \\
&\geq 3(\log(n/2) + \log(n/2) + \cdots + \log(n/2)) \\
&= 3(n/2)(\log(n/2)) \\
&= \frac{3n}{2}(\log n - \log 2) \\
&\geq \frac{3n}{4}\log n \qquad \text{for } n \geq 4 \\
&= \frac{3}{4} n \log n \\
\sum_{i=1}^{n} \log(i^3) &= \Omega(n \log n)
\end{aligned}$$

Since $\sum_{i=1}^{n} \log(i^3) = \Omega(n \log n)$ and $\sum_{i=1}^{n} \log(i^3) = O(n \log n)$, $\sum_{i=1}^{n} \log(i^3) = \Theta(n \log n)$.

Chapter 2

Data Structures and Search Algorithms

2.1 Balanced Search Tree

(§18.1, page 488) A B-tree T is a rooted tree (whose root is $T.root$) having the following properties.

1. Every node x has the following attributes.
   (a) $x.n$, the number of keys currently stored in node x,
   (b) The $x.n$ keys themselves, $x.key_1, x.key_2, \ldots, x.key_{x.n}$, stored in nondecreasing order, so that $x.key_1 \leq x.key_2 \leq \cdots \leq x.key_{x.n}$.
   (c) $x.leaf$, a boolean value that is `True` if x is a leaf and `False` if x is an internal node.
2. Each internal node x also contains $x.n + 1$ pointers, $x.c_1, x.c_2, \ldots, x.c_{x.n+1}$ to its children. Leaf nodes have no children, and so their c_i attributes are undefined.
3. The keys $x.key_i$ separate the ranges of keys stored in each subtree. If k_i is any key stored in the subtree with root $x.c_i$, then

$$k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq \cdots \leq x.key_{x.n} \leq k_{x.n+1}$$

4. All leaves have the same depth, which is the tree's height, h.
5. Nodes have lower and upper bounds on the number of keys they can contain. We express these bounds in terms of a fixed integer $t \geq 2$ called the *minimum degree* of the B-tree.
   (a) Every node other than the root must have at least $t - 1$ keys. Every internal node other than the root thus has at least t children. If the tree is nonempty, the root must have a least one key.
   (b) Every node may contain at most $2t - 1$ keys. Therefore, an internal node may have at most $2t$ children. We say that a node is *full* if it contains exactly $2t - 1$ keys.

2.1.1 Method for Inserting and Deleting a Key

In a B-tree (Balanced binary tree) with minimum degree t, each node other than the root must have at least $t-1$ keys, and each node including the root must have at most $2t-1$ keys. When we delete or insert a key, we don't know up front the node from which we're going to insert or delete the key, and we want to make just one pass from root to leaf, so we're going to adjust appropriately as we go.

When deleting a key, when passing from the root to a leaf, if a node and its children together can be merged (contain no more than $2k-1$ keys), merge them.

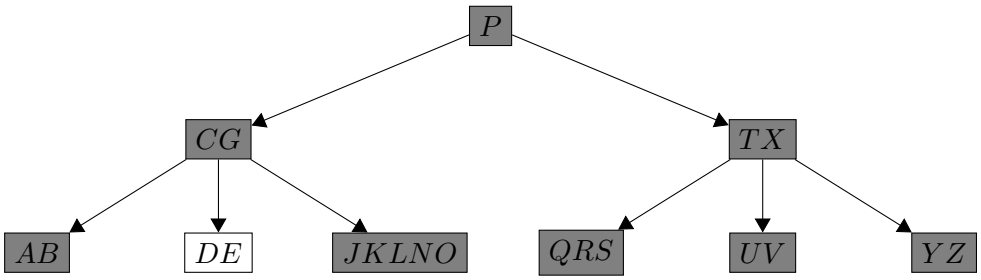
When inserting a key, when passing from the root to a leaf, if a node has the maximum $(2k-1)$ number of keys, split it.

2.1.2 Old Exam Questions

Note: Why do they introduce keys with subscripts, instead of just using A through Z? Twenty-six keys is plenty for these exercises.

S19 #L1

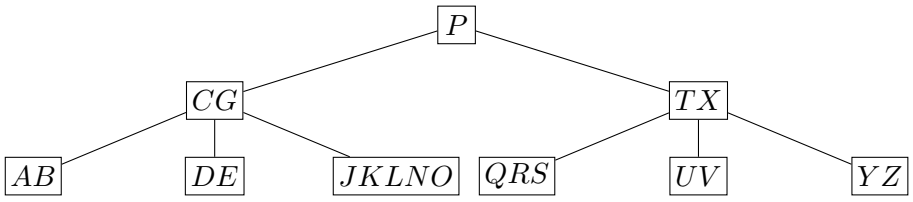
Given a B-tree with the minimum degree of $t = 3$ below, show the results after (i) deleting B, (ii) followed by inserting M, (iii) then followed by deleting T, and then inserting M_t for $M < M_t < N$



Note: The different shading of the nodes is not relevant. They copied this from some book exercise where the node color indicated some step in a process that is not described here.

Solution

Original Tree



i. Delete B.

Instructions on "Deleteing a key from a B-tree," part 3. Starting at $x = root$, "If the key k is not present in internal node x, determine the root $x.c_i$ of the appropriate subtree that must contain k, if k is in the tree at all." Since $B < P$, we must go to the left to $x.c_1 = CG$.

"If $x.c_i$ has only $t-1$ keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then finish by recursing on the appropriate child of x.

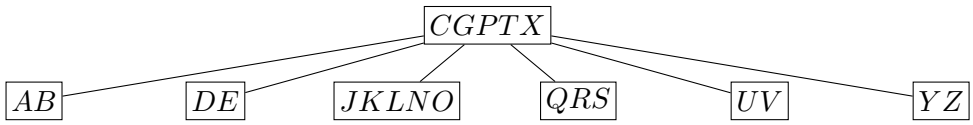
It is true that $x.c_1 = CG$ has only $t - 1 = 2$ keys.

"3a. If $x.c_i$ has only $t-1$ keys but has an immediate sibling with at least t keys,...." Not true, since the only sibling of CG is TX, which also has only $2 < t$ keys.

"3b. If $x.c_i$ and both of $x.c_i$'s immediate siblings have $t-1$ keys, merge $x.c_i$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node."

We need to merge CG and TX, moving P down into the new merged node to become the median key for that node.

First merge P, CG, and TX.



Recurse. We are at node $CGPTX$.

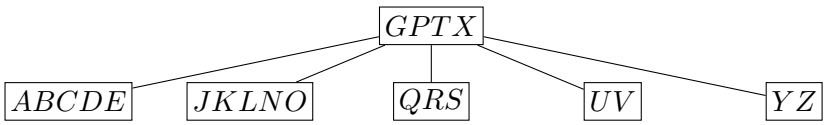
"3. If the key k is not present in internal node x, determine the root $x.c_i$ of the appropriate subtree that must contain k, if k is in the tree at all." The key B must be in the subtree rooted at $x.c_1 = AB$.

"If $x.c_1$ has only $t-1$ keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys." It is true that $x.c_1$ has only $t - 1 = 2$ keys.

"3a. If $x.c_i$ has only $t-1$ keys but has an immediate sibling with at least t keys..." Nope.

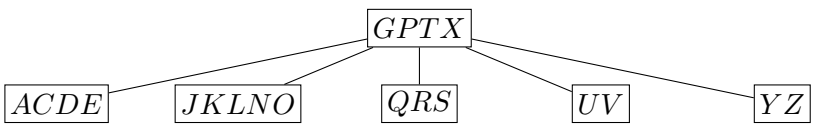
"3b. If $x.c_i$ and both of $x.c_i$'s immediate siblings have $t-1$ keys, merge $x.c_i$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node."

Merge AB and DE, bringing down C.



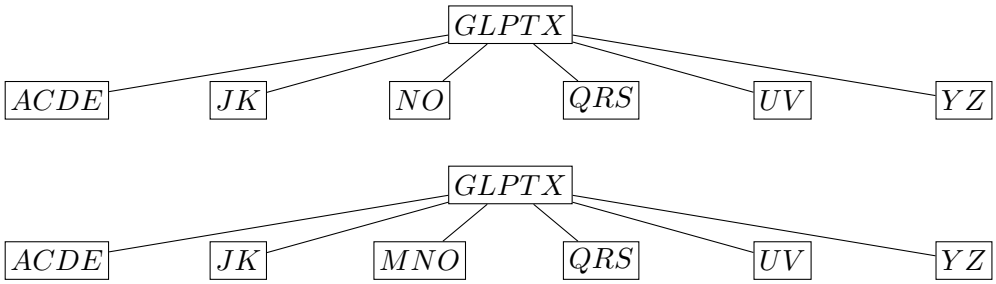
Recurse. We are at node $x = ABCDE$.

"1. If the key k is in node x and x is a leaf, delete the key k from x."



ii. Insert M.

It's going to go into $JKLNO$, but that would be too big, so split $JKLNO$ into JK and NO, moving L up into $GPTX$. Then insert M into NO.



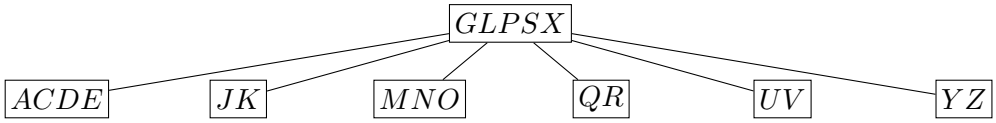
iii. Delete T.

"2. If the key k is in node x and x is an internal node, do the following:

a. If the child y that precedes k in node x has at least t keys, then find the predecessor k' of k in the subtree rooted at y. Recursively delete k', and replace k by k' in x."

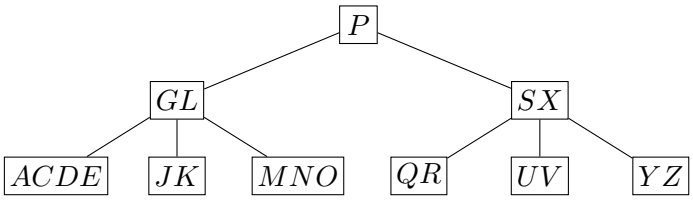
In this case, if the key T is in node $GLPTX$ and $GLPTS$ is an internal node, do the following:

a. If the child QRS that precedes T in node $GLPTX$ has at least $t = 3$ keys [which it does], then find the predecessor S of T in the subtree rooted at QRS. Recursively delete S, and replace T by S in $GLPTX$.

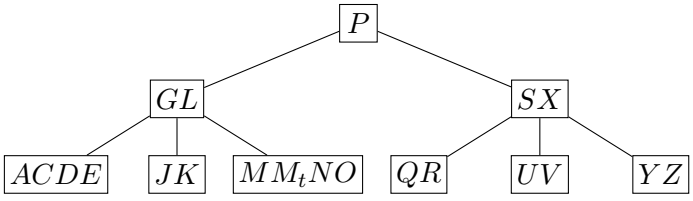


iv. Insert M_t for $M < M_t < N$.

The first thing to do is to split $GLPSX$.



Then it's just a leaf insertion.

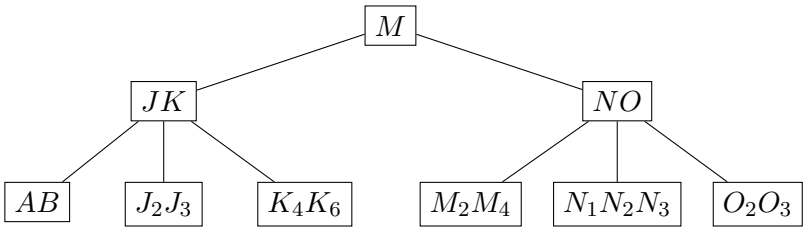


S18 #L1

Given the initial B-tree with the minimum node degree of $t = 3$ below, show the results

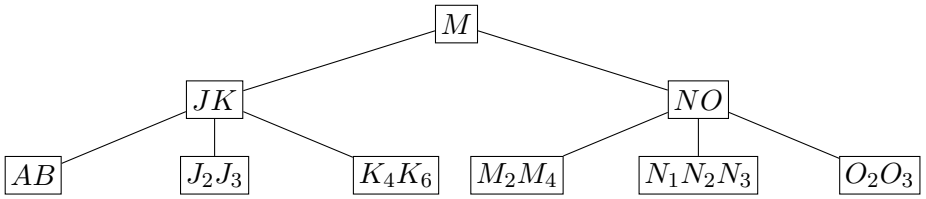
a. After deleting the key of M_2,
b. Followed by inserting the key of L,
c. Then by deleting the key of J_2,
d. Then by inserting the key of O_1 with $O < O_1 < O_2$, and
e. Then by deleting K.

(Show the result after every deletion and after every insertion.)



Solution

Original Tree



a. Delete M_2.

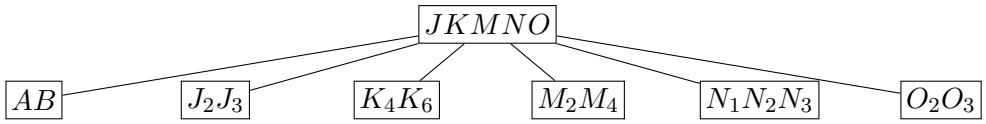
Start on node M.

Instructions on page 502.

"3. If the key k [M_2] is not present in internal node x [M] determine the node $x.c_i$ [$x.c_2 = NO$] of the appropriate subtree that must contain k [M_2], if k is in the tree at all. If $x.c_i$ has only $t - 1$ [2] keys, [TRUE], execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t [3] keys. Then finish by recursing on the appropriate child of x.

3b. If $x.c_i$ and both of $x.c_i$'s immediate siblings have $t - 1$ [2] keys, merge $x.c_i$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node."

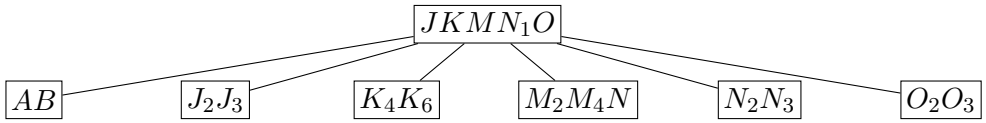
Consolidate JK, M, and NO.



"3. If the key k [M_2] is not present in internal node x [$JKMNO$] determine the node $x.c_i$ [$x.c_4 = M_2M_4$] of the appropriate subtree that must contain k [M_2], if k is in the tree at all. If $x.c_i$ has only $t-1$ [2] keys, [TRUE], execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t [3] keys. Then finish by recursing on the appropriate child of x.

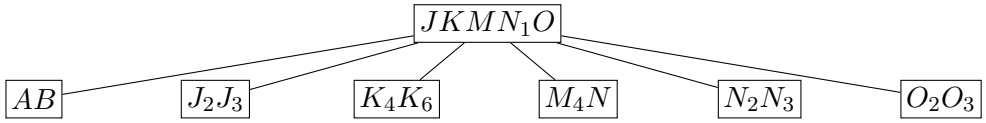
3a. If $x.c_i$ [$x.c_4 = M_2M_4$] has only $t-1$ keys but has an immediate sibling with at least t [3] keys, [TRUE], give $x.c_i$ an extra key by moving a key from x [$JKMNO$] down into $x.c_i$, moving a key from $x.c_i$'s immediate left or right sibling up into x, and moving the appropriate child pointer [N] from the sibling into $x.c_i$."

Move N_1 into $JKMNO$, then move N down into M_2M_4.

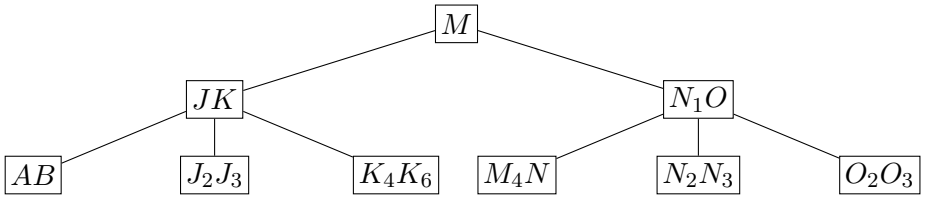


Recurse to node M_2M_4N.

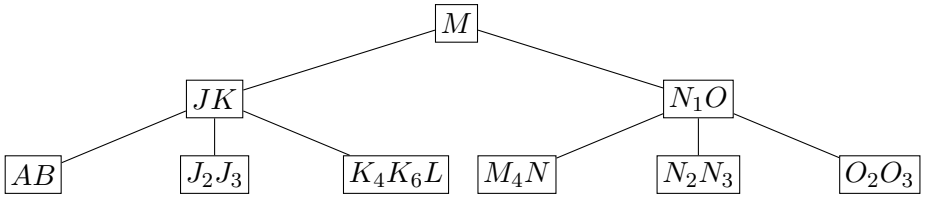
"1. If the key k [M_2] is in node x and x is a leaf, [TRUE], delete the node k from x.



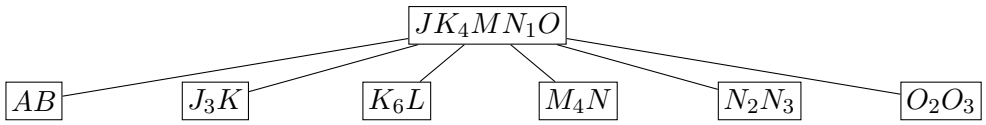
b. Insert L**.** Start by splitting $JKMN_1O$, because it has maximum size and we wouldn't be able to insert something into it if necessary.



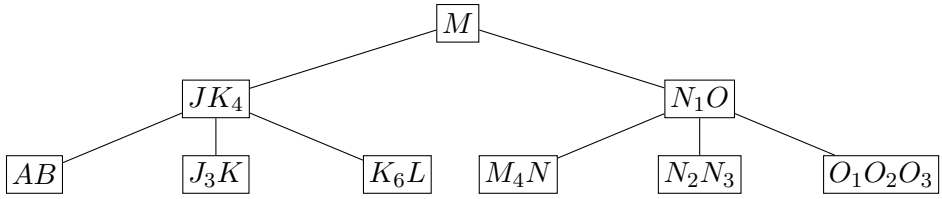
Then it's a leaf insert.



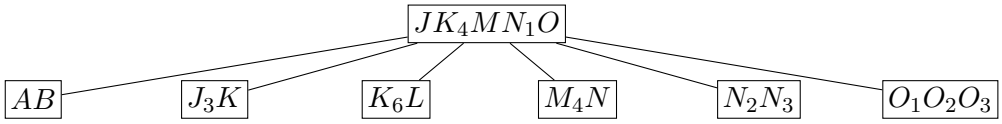
c. Delete J_2. Re-merge JK, M, and N_1O. Then rotate K and K_4 around to have the minimum number of keys in each node.



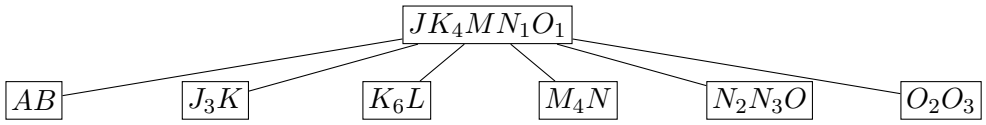
c. Insert O_1 with $O < O_1 < O_2$. Re-split JK_4MN_1O, then simple leaf insertion.



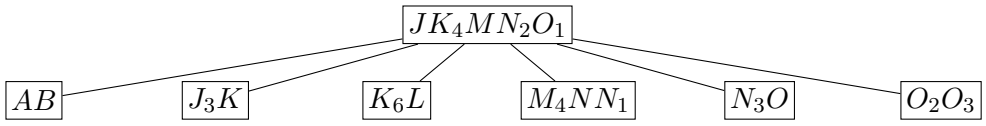
d. Delete K. Start with re-merging JK_4, M, and N_1O. Lots of rotation.



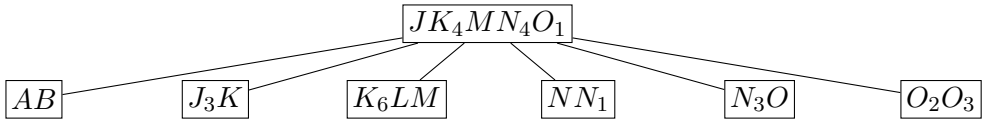
Rotate O and O_1 left.



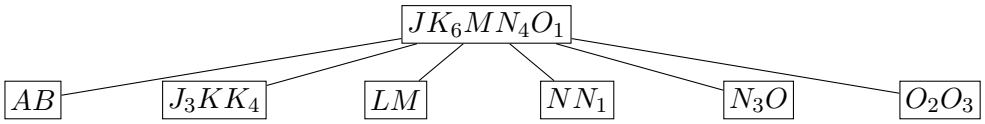
Rotate $N1$ and N_2 left.



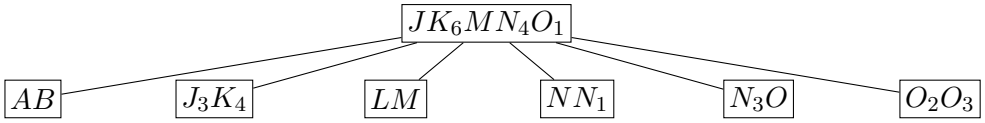
Rotate M and M_4 left.



Rotate K_4 and K_6 left.



Delete K from the node.

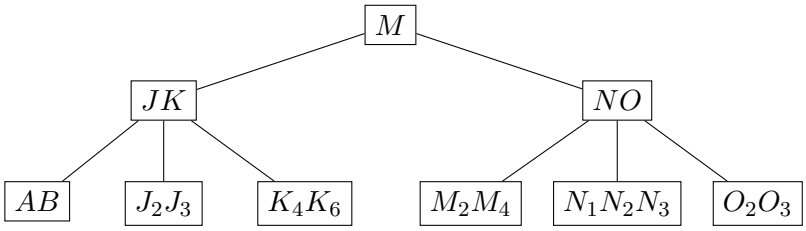


F18 #L2

Given the initial B-tree with the minimum node degree of $t = 3$ below, show the results

a. After deleting the key of K,
b. Followed by inserting the key of L,
c. Then by deleting the key of J_2,
d. Then by inserting the key of N_4 with $N_3 < N_4 < O$, and
e. Then by deleting K_4.

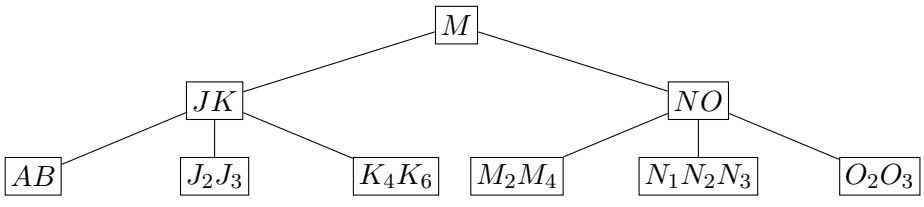
(Show the result after every deletion and after every insertion.)



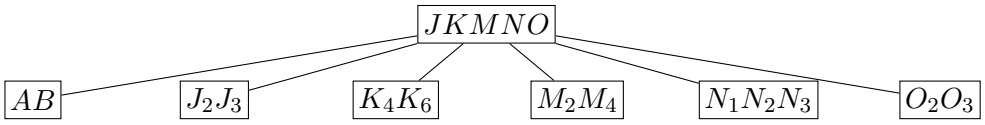
Note: Same tree as in S18 #L1, but different keys to insert and delete.

Solution

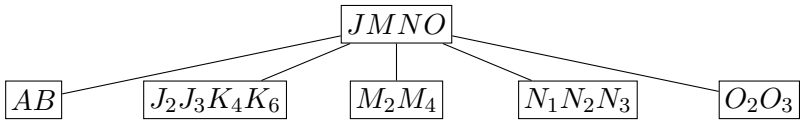
Original Tree



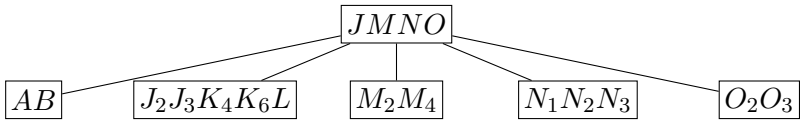
a. Delete K. First, merge JK, M, and NO.



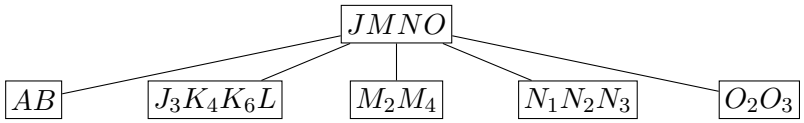
Then delete K and merge J_2J_3 and K_4K_6.



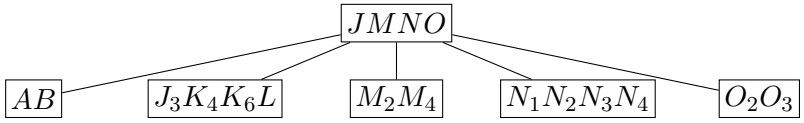
b. Insert L. Just a simple leaf insert.



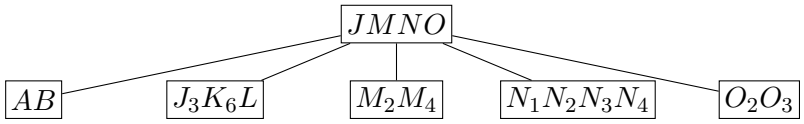
c. Delete J_2. Just a simple leaf deletion.



d. Insert N_4 with $N_3 < N_4 < O$. Just a simple leaf insertion.



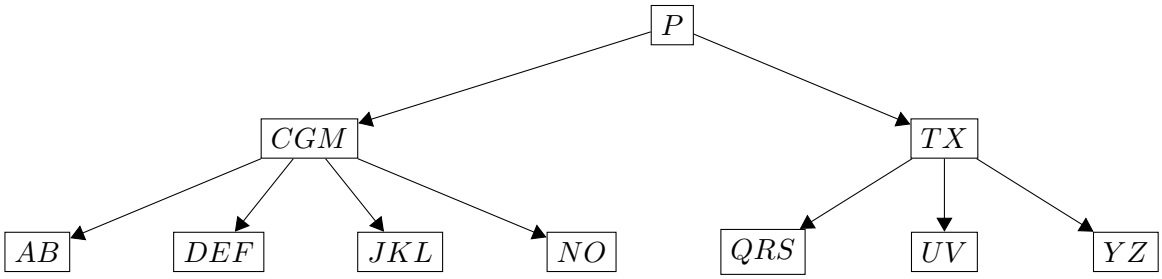
e. Delete K_4. Just a simple leaf deletion.



F16 #L3

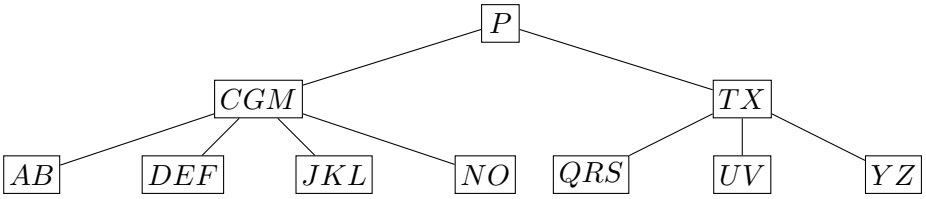
Given an initial B-tree with the minimum node degree of $t = 2$ below, show the results

1. after inserting the key of H, and
2. then followed by deleting two keys in order: X then P. (show the result after insertion and the result after each deletion.)

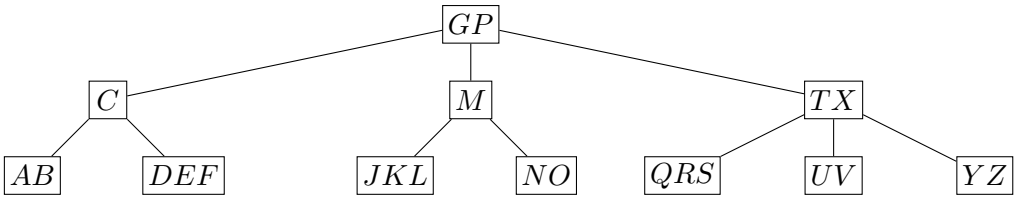


Solution

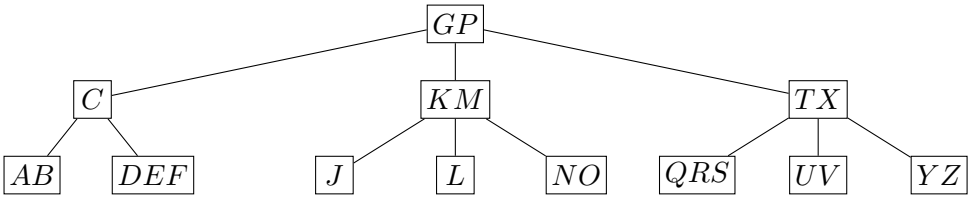
Original Tree



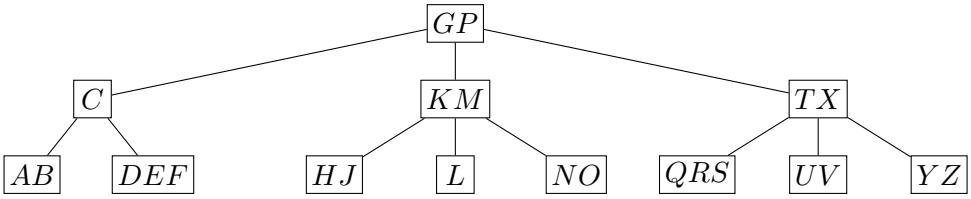
Insert H. As we go down the tree toward where H should go, we encounter a full node, CGM, so we have to split it.



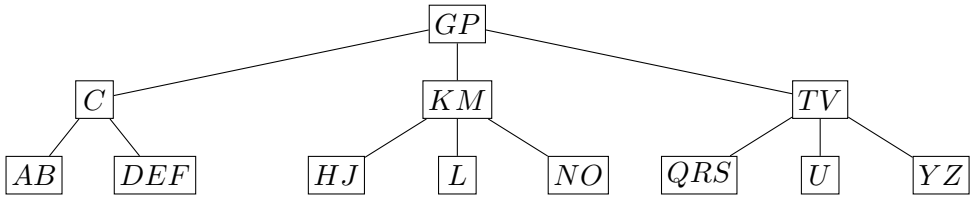
Then going down we want to insert H in JKL, but it is full, so we have to split it.



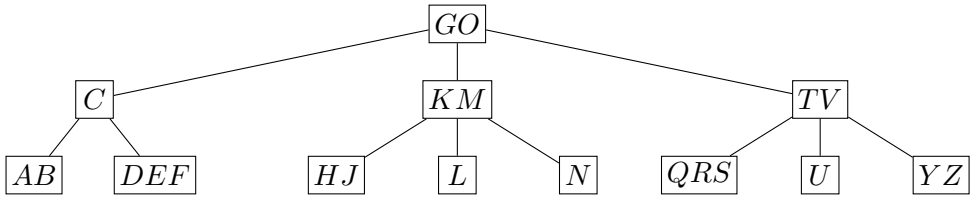
Now we can insert H.



Delete X. Move V up.



Delete P. Move O up.



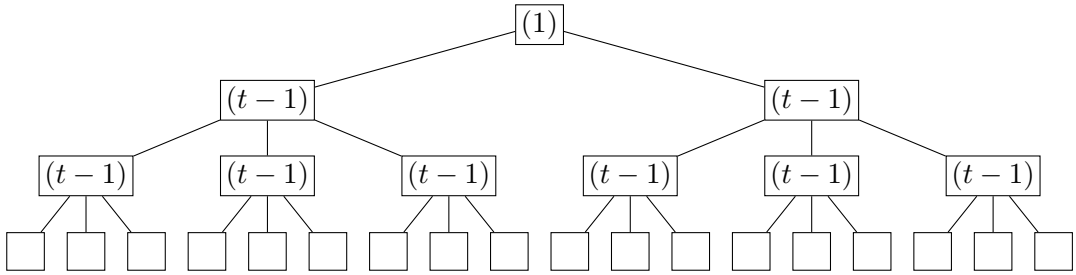
F15 #S8

For any n-key B-tree of height h and with minimum node degree of $t \geq 2$, prove that h is no larger than $\log_t \frac{n+1}{2}$. (Hint: Consider the number of keys stored in each tree level.)

Solution

For illustration, take $t = 3$. In each node is the minimum number of keys in the node.

We're interested in the *minimum* number of keys per node, so that we get the maximum possible height for n keys, giving us an upper bound for the height of the tree.



The root node (depth 0) has at least one key; thus, two children. At depth 1, each of the two nodes has at least $t - 1$ keys; thus, t children. At any greater depth, each node has at least $t - 1$ keys; thus, t children.

	Minimum Number of	
Depth	Nodes	Keys
0	1	1
1	2	$2(t-1)$
2	$2t$	$2t(t-1)$
3	$2t^2$	$2t^2(t-1)$
	$\vdots$	
h	$2t^{h-1}$	$2t^{h-1}(t-1)$

$$
\begin{aligned}
1 + 2(t-1) + 2t(t-1) + 2t^2(t-1) + \cdots + 2t^{h-1}(t-1) &= n \\
2(t-1)(1 + t + t^2 + \cdots + t^{h-1}) &= n-1 \\
(t-1)\left(\frac{t^h - 1}{t-1}\right) &= \frac{n-1}{2} \\
t^h - 1 &= \frac{n-1}{2} \\
t^h &= \frac{n-1}{2} + 1 \\
t^h &= \frac{n+1}{2} \\
h &= \log_t \frac{n+1}{2}
\end{aligned}
$$

Related Question

For any n-key B-tree of height h and with minimum node degree of $t \geq 2$, find the minimum value of h, the height of the tree.

	Maximum Number of	
Depth	Nodes	Keys
0	1	$2t-1$
1	$2t$	$2t(2t-1)$
2	$(2t)^2$	$(2t)^2(2t-1)$
	$\vdots$	
h	$(2t)^h$	$(2t)^h(2t-1)$

$$
\begin{aligned}
(2t-1) + 2t(2t-1) + (2t)^2(2t-1) + \cdots (2t)^h(2t-1) &= n \\
(2t-1)(1 + (2t) + (2t)^2 + (2t)^h) &= n \\
(2t-1)\left(\frac{(2t)^{h+1}-1}{2t-1}\right) &= n \\
(2t)^{h+1} - 1 &= n \\
(2t)^{h+1} &= n+1 \\
h+1 &= \log_{2t}(n+1) \\
h &= \log_{2t}(n+1) - 1
\end{aligned}
$$

For any n-key B-tree of height h and minimum node degree of $t \geq 2$,

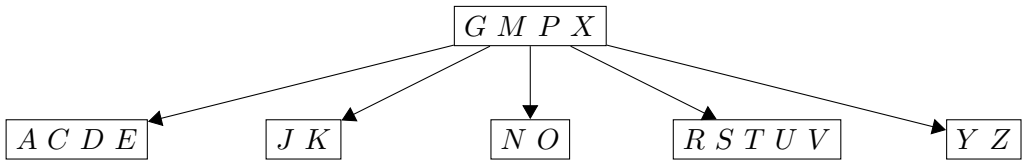
$$\log_{2t}(n+1) - 1 \leq h \leq \log_t \frac{n+1}{2}$$

For instance, if $t = 2$ and $n = 63$, then the height is between 2 and 5.

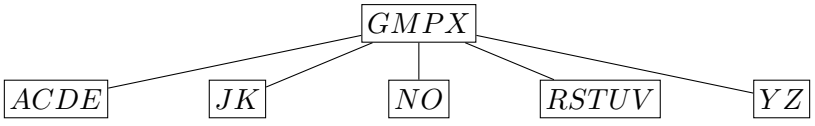
$$\log_{2t}(n+1) - 1 = \log_4 64 = 3, \qquad \log_t \frac{n+1}{2} = \log_2 32 = 5$$

F15 #L2

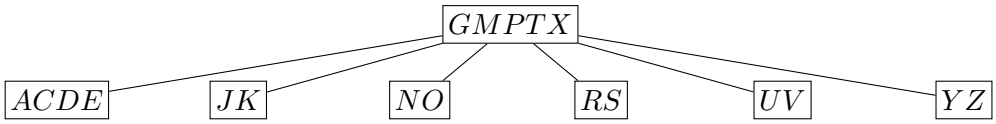
Given the initial B-tree with the minimum node degree of $t = 3$ below, show the results (a) after inserting two keys in order: Q then W, and (b) followed by deleting two keys in order: Y then T. (Show the aggregate result after insertion and another result after deletion.)



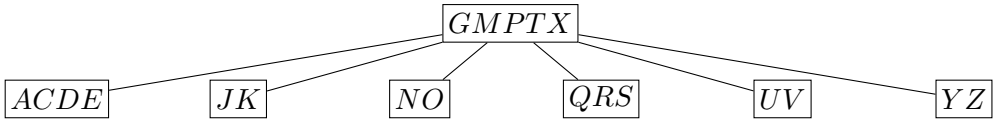
Solution



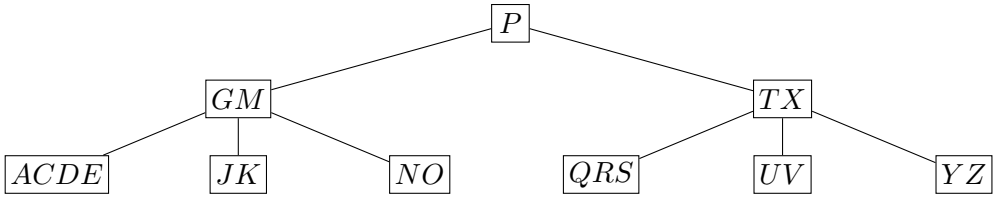
To insert Q, we note that $RSTUV$ is full, because $x.n = 5 = 2t - 1$, so we must split it by moving T up.



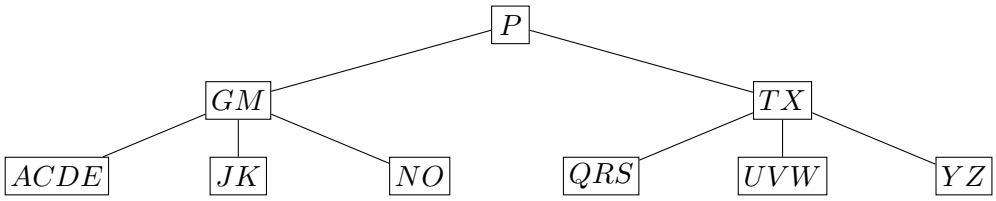
Then insert Q into a leaf, simple leaf insertion.



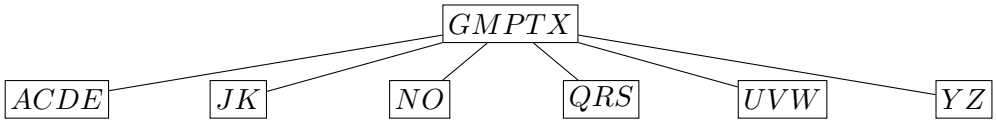
When we go to make another insertion, we hit a full node, $GMPSX$, so we must split it first.



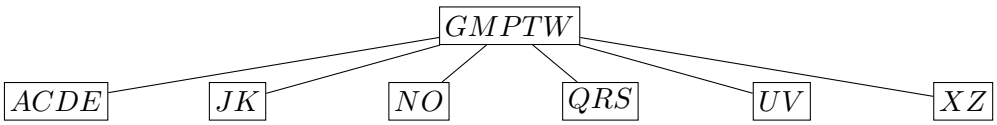
Now W is a simple leaf insertion.



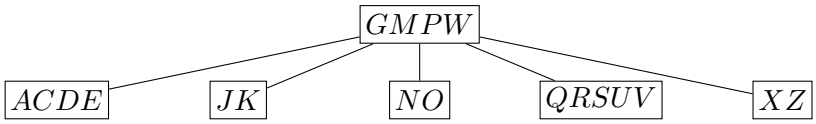
When we want to delete a key (Y), going down the graph we see that we can re-merge GM, P, and TX.



Now delete Y by rotating W and X around to the right.



To delete T, just merge QRS and UV.



2.2 Binary Search Trees

2.2.1 Definition, Chapter 12

The linked data structure in which every node is an object has a *key* and attributes $left$, $right$, and p that point to the nodes of the left child, right child, and parent. If the node does not have such an attribute, its value is `NIL`. The root node is the only node whose parent is `NIL`.

Binary Search Tree Property. Let x be a node in a binary search tree. If y is a node in the left subtree of x, then $y.key \leq x.key$. If y is a node in the right subtree of x, then $x.key \leq y.key$.

Note that it doesn't just apply to the children of x, but any nodes in the subtree rooted at x.

Operations, with n being the number of nodes and h being the height of the tree, $\log_2 n \leq h \leq n$. The best case for h is that the tree is balanced, with $h = \log_2 n$, and the worst case is that the tree is a linear chain of n nodes, $h = n$.

Inorder-Tree-Walk	$\Theta(n)$
Tree-Search	$O(n)$
Tree-Minimum	$O(h)$
Tree-Maximum	$O(h)$
Tree-Successor	$O(h)$
Tree-Predecessor	$O(h)$
Tree-Insert	$O(h)$
Tree-Delete	$O(h)$

2.2.2 Old Exam Questions

S15 #S9

Mark true/false against the following statements.

a. (Same as F15 #S7a) A binary search tree of size N will always find a key in at most $O(\log N)$ time.
b. (Same as F15 #S5c) A breadth first search algorithm can be considered as a special case of heuristic search algorithm.
c. (Same as F15 #Sb) An optimal binary search tree is not necessarily a balanced tree.
d. A dynamic programming approach uses top-down problem solving strategy to solve optimization problem.

Solutions

a. False. A binary search tree of height h will always find a key in at most $O(h)$ time. In a *balanced* binary search tree, the height is about $\log N$, so the statement is true for balanced trees, but a binary search tree can also be a linear chain with height N, so the search could take up to $O(N)$ time.
b. No idea. We didn't cover heuristics, and it's not in the textbook.
c. True.
d. True. One dynamic programming approach uses a top-down problem solving with memoization strategy, and another uses bottom-up problem solving.

F15 #S7

Parts (a) and (b) the same as (a) and (c) above.

Mark True or False against the following statements.

a. (Same as S15 #9a) A binary search tree of size N will always find a key in at most $O(\log N)$ time.
b. An optimal binary search is not necessary a balanced tree.
c. A binary heap always maintains a balanced tree as practical as it can be.
d. To implement a priority queue binomial heap is preferred over binary heap.
e. A graph formed by strongly connected components, a strongly connected components graph (SCC) is always a minimum spanning tree.

Solutions

a. False. If the tree is balanced, yes. If not, it could be up to $O(n)$ time.
b. True.
c. True-ish. A binary heap is a balanced tree. Not "as practical as it can be." Just is.
d. No idea. Not in our textbook. "Binomial heap" is in an exercise, but no priority queue binomial heap.
e. False. A strongly connected graph is a spanning tree, but not necessarily a minimal spanning tree.

F16 #S8, First part, Same as above

Mark true/false (T/F) against the following statements.

[Note that the subquestions were not enumerated in the original. I enumerated them so I could write the solutions and justifications separately.]

1. A binary search tree of size N will always find a key at most $O(\lg N)$ time
2. A breadth first search can be considered as a special case of heuristic search algorithm.
3. An optimal binary search tree is not necessarily being a balanced tree.

4. A dynamic programming approach uses top-down problem solving strategy to solve optimization problem.

Solutions

1. False. A balanced binary search tree will always find a tree in at most $O(\lg N)$ time, but a binary search tree does not have to be balanced. In the worst case, if the tree is just one branch, it would require $O(N)$ time.
2. No idea. This book doesn't cover heuristics, and we didn't talk about it in class.
3. True.
4. True, if the question means, "There is a dynamic programming approach that uses a top-down decision strategy..."
   False if the question means, "A dynamic programming approach must use a top-down problem solving stragegy..." or "The dynamic programming approach uses the top-down problem solving strategy..."
   The question statement is missing an article preceding "top-down programming approach," so I will not trust my grade to interpretations of the subtleties of the use of another article in the sentence.
   Dynamic programming can use either top-down with memoization or bottom-up.

2.3 Optimal Binary Search Tree

2.3.1 Definition, §15.5

A tree whose expected search cost is smallest.

- A set of n *distinct* keys in sorted order, $K = (k_1, k_2, \ldots, k_n)$ with $k_1 < k_2 < \cdots < k_n$
- For each key k_i, a probability p_i that a search will be for k_i.
- A set of $n+1$ dummy keys $d_0, d_1, \ldots d_n$ representing values not in K, such that d_0 represents anything less than k_1, d_1 represents anything between k_1 and k_2, ..., and d_n represents anything greater than k_n.
- For each key d_i, a probability q_i that the search will correspond to d_i.
- Each d node is a leaf; each k node is internal.

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

- Given a binary search tree T, the cost of a search is the number of nodes examined, which is one more than the depth of the node found by the search.

$$
\begin{aligned}
E[\text{Search cost in } T] &= \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\text{depth}_T(d_i) + 1) \cdot q_i \\
&= \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=1}^{n} p_i + \sum_{i=0}^{n} \text{depth}_T(d_i) \cdot q_i + \sum_{i=0}^{n} q_i \\
&= 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \text{depth}_T(d_i) \cdot q_i
\end{aligned}
$$

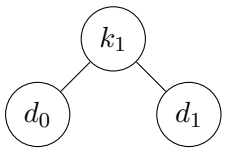
2.3.2 Brad's Example

Unlike the examples in the text, I made all of the probabilities different, making sure their sum is 1.00.

i	0	1	2	3	4	5
p_i		0.10	0.11	0.12	0.13	0.15
q_i	0.04	0.05	0.06	0.07	0.08	0.09

w	0	1	2	3	4	5
1	0.04	0.19	0.36	0.55	0.76	1.00
2		0.05	0.22	0.41	0.62	0.86
3			0.06	0.25	0.46	0.70
4				0.07	0.28	0.52
5					0.08	0.32
6						0.09

There is only one way to construct the subtree containing only one of the k nodes.

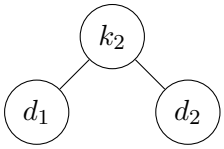


$1 \times (p_1) + 2 \times (q_0 + q_1)$
$= (q_0) + (q_1) + (p_1 + q_0 + q_1)$
$= e[1,0] + e[2,1] + w[1,1]$
$= 0.28 \rightarrow e[1,1]$
$1 \rightarrow root[1,1]$

$w[1,1] = 0.19$

e	0	1	2	3	4	5
1	0.04	0.28				
2		0.05				
3			0.06			
4				0.07		
5					0.08	
6						0.09

$root$	1	2	3	4	5
1	1				



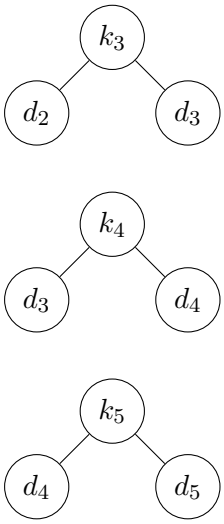
$1 \times (p_2) + 2 \times (q_1 + q_2)$
$= (q_1) + (q_2) + (p_2 + q_1 + q_2)$
$= e[2,1] + e[3,2] + w[2,2]$
$= 0.33 \rightarrow e[2,2]$
$2 \rightarrow root[2,2]$

$w[2,2] = 0.19$

e	0	1	2	3	4	5
1	0.04	0.28				
2		0.05	0.33			
3			0.06			
4				0.07		
5					0.08	
6						0.09

$root$	1	2	3	4	5
1	1				
2		2			

...and so forth.

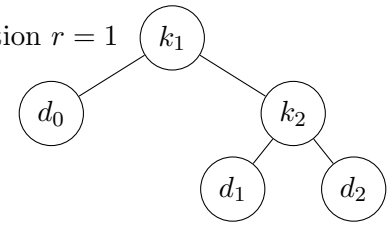


e	0	1	2	3	4	5
1	0.04	0.28				
2		0.05	0.33			
3			0.06	0.38		
4				0.07	0.43	
5					0.08	0.49
6						0.09

$root$	1	2	3	4	5
1	1				
2		2			
3			3		
4				4	
5					5

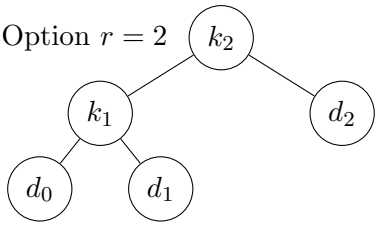
There are two ways to construct each of the subtrees containing two k nodes. Note how each of these subtrees contains one of the above subtrees. Choose the option that gives the optimal subtree, *i.e* with the lowest expected cost.

$e[1,2]$, Option $r = 1$



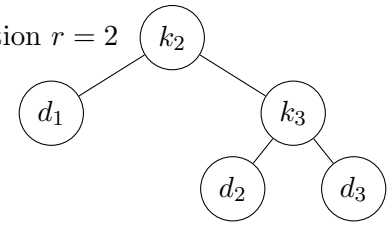
$$1 \times (p_1) + 2 \times (q_0 + p_2) + 3 \times (q_1 + q_2)$$
$$(q_0) + (p_2 + 2(q_1 + q_2)) + (p_1 + p_2 + q_0 + q_1 + q_2)$$
$$e[1,0] + e[2,2] + w[1,2]$$
$$= 0.73$$

$e[1,2]$, Option $r = 2$



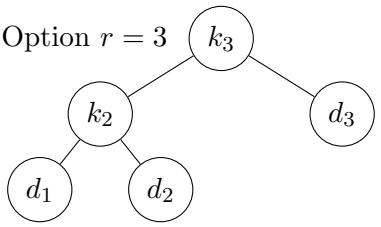
$$1 \times (p_2) + 2 \times (p_1 + q_2) + 3 \times (q_0 + q_1)$$
$$(p_1 + 2(q_0 + q_1)) + (q_2) + (p_1 + p_2 + q_0 + q_1 + q_2)$$
$$e[1,1] + e[3,2] + w[1,2]$$
$$0.70 \rightarrow e[1,2] \text{ Optimal, } \; 2 \rightarrow root[1,2]$$

$e[2,3]$, Option $r = 2$



$$1 \times (p_2) + 2 \times (q_1 + p_3) + 3 \times (q_2 + q_3)$$
$$(q_1) + (p_3 + 2(q_2 + q_3)) + (p_2 + p_3 + q_1 + q_2 + q_3)$$
$$e[2,1] + e[3,3] + w[2,3]$$
$$= 0.84$$

$e[2,3]$, Option $r = 3$



$$1 \times (p_3) + 2 \times (p_2 + q_3) + 3 \times (q_1 + q_2)$$
$$(p_2 + 2(q_1 + q_2)) + (q_3) + (p_2 + p_3 + q_1 + q_2 + q_3)$$
$$e[2,2] + e[4,3] + w[2,3]$$
$$0.81 \rightarrow e[3,2] \text{ Optimal, } \; 3 \rightarrow root[3,2]$$

e	0	1	2	3	4	5
1	0.04	0.28	0.70			
2		0.05	0.33	0.81		
3			0.06	0.38	0.92	
4				0.07	0.43	1.04
5					0.08	0.49
6						0.09

$root$	1	2	3	4	5
1	1	2			
2		2	3		
3			3	4	
4				4	5
5					5

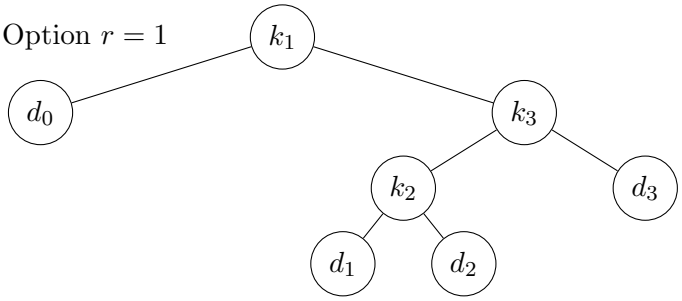
There are five ways to make a subtree with k_1, k_2, and k_3, but we already know that two of them are poor choices.

- Rooted at k_1.
  - Put subgraph 2.C above as the right subgraph of k_1.

– Put subgraph 2.D above as the right subgraph of k_1.

Either option has the same overhead, pushing k_2, k_3, d_1, d_2, and d_3 down a level and adding k_1 and level 0 and d_0 at level 1, but 2.D has a lower expected cost than 2.C, so we do not need to consider 2.C.

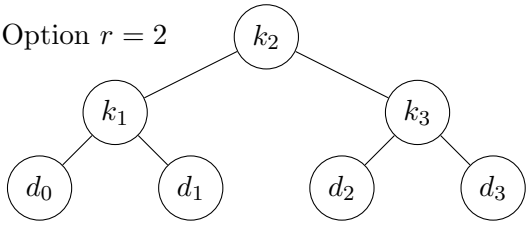
$e[1,3]$, Option $r = 1$



$$1 \times (p_1) + 2 \times (q_0 + p_3) + 3 \times (p_2 + q_3) + 4 \times (q_1 + 1_2)$$
$$(q_0) + (p_3 + 2(p_2 + q_3) + 3(q_1 + q_2)) + (p_1 + p_2 + p_3 + q_0 + q_1 + q_2 + q_3)$$
$$e[1,0] + e[2,3] + w[1,3]$$
$$= 1.40$$

- Rooted at k_2. This option has two single-k subgraphs.

$e[1,3]$, Option $r = 2$

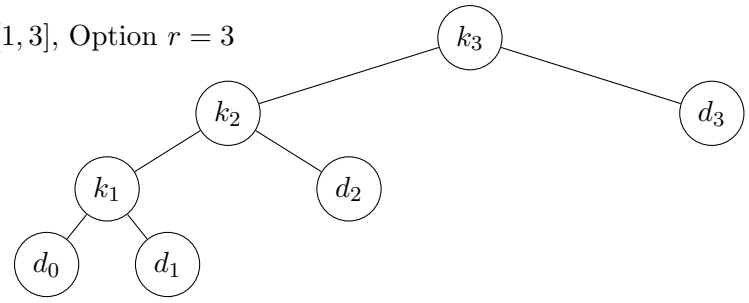


$$1 \times (p_2) + 2 \times (p_1 + q_3) + 3 \times (q_0 + q_1 + q_2 + q_3)$$
$$(p_1 + 2q_0 + 2q_1) + (p_3 + 2q_2 + 2q_3) + (p_1 + p_2 + p_3 + q_0 + q_1 + q_2 + q_3)$$
$$e[1,1] + e[3,3] + w[1,3]$$
$$= 1.21 \rightarrow e[3,1] \text{ Optimal}, \;\; 2 \rightarrow root[1,3]$$

- Rooted at k_3.
  - Put subgraph 2.A above as the left subgraph of k_3.
  - Put subgraph 2.B above as the left subgraph of k_3.

Either option adds the same overhead of pushing k_1, k_2, d_0, d_1, and d_2 down a level, putting k_3 at level 0, and putting d_3 at level 1, thus adding $p_1 + p_2 + q_0 + q_1 + q_2 + p_3 + 2 \cdot q_3$ to the expected cost of the existing subgraph. Since 2.B has a lower expected cost than 2.A, we do not need to consider 2.A.

$e[1,3]$, Option $r = 3$



$$1 \times (p_3) + 2 \times (p_2 + q_3) + 3 \times (p_1 + q_2) + 4 \times (q_0 + q_1)$$
$$(2p_1 + p_2 + 3q_0 + 3q_1 + 2q_2) + (q_3) + (p_1 + p_2 + p_3 + q_0 + q_1 + q_2 + q_3)$$
$$e[1,2] + e[4,3] + w[1,3]$$
$$= 1.32$$

Continue adding layers.

e	0	1	2	3	4	5
1	0.04	0.28	0.70	1.21	1.89	2.70
2		0.05	0.33	0.81	1.38	2.16
3			0.06	0.38	0.92	1.57
4				0.07	0.43	1.04
5					0.08	0.49
6						0.09

w	0	1	2	3	4	5
1	0.04	0.19	0.36	0.55	0.76	1.00
2		0.05	0.22	0.41	0.62	0.86
3			0.06	0.25	0.46	0.70
4				0.07	0.28	0.52
5					0.08	0.32
6						0.09

$root$	1	2	3	4	5
1	1	2	2	3	4
2		2	3	3	4
3			3	4	4
4				4	5
5					5

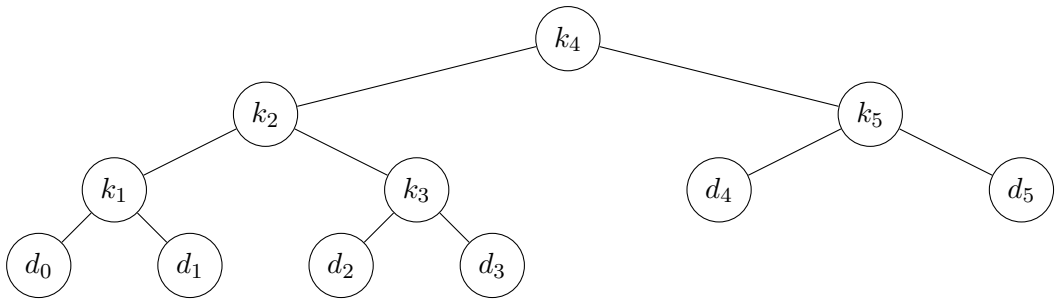
Here's how to build the structure of the optimal binary tree from $root$.

The element $root[1,5] = 4$ indicates that the subgraph of $k_1, \ldots, k_5$, *i.e.* the entire graph, has k_4 as its root.

If k_4 is the root, then k_5 is its right child, and its left child is the optimal subtree containing

k_1, k_2, and k_3. The element $root[1,3] = 2$ indicates that k_2 is the root of this subtree, implying that k_1 is its left child and k_3 is its right.

Fill in the d_i's.



$$\begin{aligned}
e[1,5] &= 1 \times (p_4) + 2 \times (p_2 + p_5) + 3 \times (p_1 + p_3 + q_4 + q_5) + 4 \times (q_0 + q_1 + q_2 + q_3) \\
&= 1(0.13) + 2(0.11 + 0.15) + 3(0.10 + 0.12 + 0.08 + 0.09) + (0.04 + 0.05 + 0.06 + 0.07) \\
&= 1(0.13) + 2(0.26) + 3(0.39) + 4(0.22) \\
&= 0.13 + 0.52 + 1.17 + 0.88 \\
&= 2.70 \ \surd
\end{aligned}$$

Here's the code that takes in n, p, and q and outputs the three tables in LaTeX format.

```
import math


#n = 5
#p = [0.00, 0.10, 0.11, 0.12, 0.13, 0.15]
#q = [0.04, 0.05, 0.06, 0.07, 0.08, 0.09]

n = 7
p = [1/n for i in range (n+1)]
p[0] = 0
q = [0 for i in range (n+1)]

print (sum(p) + sum(q))

e = [[0.0 for y in range (n+1)] for x in range (n+2)]
w = [[0.0 for y in range (n+1)] for x in range (n+2)]
root = [[0.0 for y in range (n+1)] for x in range (n+1)]
```

```
for i in range (1,n+2):
    e[i][i-1] = q[i-1]
    w[i][i-1] = q[i-1]

for l in range (1,n+1):
    for i in range (1, n-l+2):
        j = i+l-1
        e[i][j] = 100.0
        w[i][j] = w[i][j-1] + p[j] + q[j]
        for r in range (i, j+1):
            t = e[i][r-1] + e[r+1][j] + w[i][j]
            if t < e[i][j]:
                e[i][j] = t
                root[i][j] = r
        print (l, i, j)
        for a in range (1,n+2):
            for b in range (0, n+1):
                print ("%3.2f  " % (e[a][b]), end='')
            print ("\\cr")
        print ()
        for a in range (1,n+2):
            for b in range (0, n+1):
                print ("%3.2f " % (w[a][b]), end='')
            print ("\\cr")
        print ()
        for a in range (1,n+1):
            for b in range (1, n+1):
                print ("%d  " % (root[a][b]), end='')
            print ("\\cr")
        print ()


for i in range (1,n+2):
    print (i, end='')
    for j in range (0,n+1):
        print (" & ", end='')
        if e[i][j]>0:
            print ("%3.2f " % (e[i][j]), end='')
    print (" \\cr")
```

```
print ()

for i in range (1,n+2):
    print (i, end='')
    for j in range (0,n+1):
        print ("␣&␣", end='')
        if w[i][j]>0:
            print ("%3.2f␣" % (w[i][j]), end='')
    print ("␣\\cr")
print ()

for i in range (1,n+1):
    print (i, end='')
    for j in range (1,n+1):
        print ("␣&␣", end='')
        if root[i][j]>0:
            print ("%d␣" % (root[i][j]), end='')
    print ("␣\\cr")
print ()
```

2.3.3 Old Exam Questions

S15 #S9, Part (c)

Mark true/false against the following statements.

a. (Same as F15 #S7a) A binary search tree of size N will always find a key in at most $O(\log N)$ time.
b. (Same as F15 #S5c) A breadth first search algorithm can be considered as a special case of heuristic search algorithm.
c. An optimal binary search tree is not necessarily a balanced tree.
d. A dynamic programming approach uses top-down problem solving strategy to solve optimization problem.

Solution to Part (c)

True. Minimizing the total expected cost may (probably will) require rotating things around to put the nodes of higher probability higher in the tree.

F15 #S7, Part (b), Same as above

Mark True or False against the following statements.

a. (Same as S15 #9a) A binary search tree of size N will always find a key in at most $O(\log N)$ time.
b. An optimal binary search is not necessary a balanced tree.
c. A binary heap always maintains a balanced tree as practical as it can be.
d. To implement a priority queue binomial heap is preferred over binary heap.
e. A graph formed by strongly connected components, a strongly connected components graph (SCC) is always a minimum spanning tree.

F16 #S8, Third part, Same as above

Mark true/false (T/F) against the following statements.

- A binary search tree of size N will always find a key at most $O(\lg N)$ time
- A breadth first search can be considered as a special case of heuristic search algorithm.
- An optimal binary search tree is not necessarily being a balanced tree.
- A dynamic programming approach uses top-down problem solving strategy to solve optimization problem.

S15 #L4

a. An object r is accessed by its key k_r. If all the objects have an equal chance of being accessed, what data structure will help you to have a better tight bound? (State your assumptions and provide the bound you have obtained for the proposed structure.)
b. An optimal binary search tree (`OPTIMAL-BST`) for a given set of keys with known access probabilities ensures the minimum expected search cost for key accesses, with its pseudo code listed below. Given the set of three keys with their access probabilities $k_1 = 0.25$, $k_2 = 0.15$, $k_3 = 0.3$, respectively, and four non-existing probabilities of $d_0 = 0.1$, $d_1 = 0.05$, $d_2 = 0.08$, $d_3 = 0.07$, construct optimal BST following dynamic programming with memorization for the given three keys and demonstrate the constructed optimal BST, which contains all three keys (k_1, k_2, k_3) and four non-exististing dummies (d_0, d_1, d_2, d_3). (Show your work using the three tables for expected costs, $e[i, j]$, for access weights, $w[i, j]$, and for $root[i, j]$, with i in $e[i, j]$ and $w[i, j]$ ranging from 1 to 4, j in $e[i, j]$ and $w[i, j]$ ranging from 0 to 3, and both i and j in $root[i, j]$ ranging from 1 to 3.)

```
OPTIMAL-BST(p,q,n)
1  let e[1..n+1,0..n], w[1..n+1,0..n], and root[1..n,1..n] be new tables.
2  for i = 1 to n+1
3          e[i,i-1] = q_{i-1}
4          w[i,i-1] = q_{i-1}
5  for l = 1 to n
6          for i = 1 to n-l+1
7                  j = i+l-1
```

```
 8                    e[i, j] = ∞
 9                    w[i, j] = w[i, j − 1] + p_j + q_j
10                    for r = i to j
11                            t = e[i, r − 1] + e[r + 1, j] + w[i, j]
12                            if t < e[i, j]
13                                    e[i, j] = t
14                                    root[i, j] = r
15  return e and root
```

[Editor's Note] This question seems like it's supposed to be accessible (if tedious) to people who haven't worked with optimal binary search trees before, but it doesn't tell you what p and q are. It should either say that $p_i = k_i$ and $q_i = d_i$, or it should use p and q instead of k and d.

Solution to (a) TO DO

Balanced binary search tree?

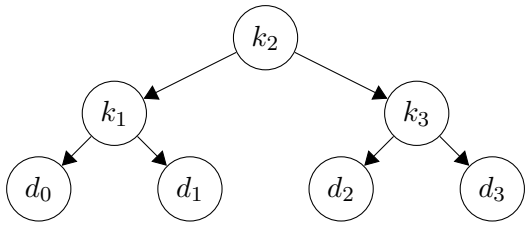
Solution to (b)

e	0	1	2	3
1	0.10	0.55	1.14	2.15
2	0.00	0.05	0.41	1.13
3	0.00	0.00	0.08	0.60
4	0.00	0.00	0.00	0.07

w	0	1	2	3
1	0.10	0.40	0.63	1.00
2	0.00	0.05	0.28	0.65
3	0.00	0.00	0.08	0.45
4	0.00	0.00	0.00	0.07

root	1	2	3
1	1	1	2
2	0	2	3
3	0	0	3

Here's how to read the `root` array. The element `root[1][3] = 2` is the node at the root of the tree containing k_1, k_2, and k_3.



$1 \times (0.15)$

$2 \times (0.25 + 0.3)$

$3 \times (0.1 + 0.05 + 0.08 + 0.07)$

$0.15 + 1.10 + 0.9 = 2.15 = e[1][3]$

2.3.4 F16 #L4, Similar to above

[Editor's Note: In the copy that I have of the exam, the probabilities were blacked out.]

Given a set of 4 keys, with the following probabilities, determine the cost and the structure of an optimal binary search tree, following the tabular, bottom-up method realized in the procedure of `OPTIMAL-BST` below to construct and fill tables $e[1..5, 0..4]$, $w[1..5, 0..4]$, and $root[1..4, 1..4]$.

i	0	1	2	3	4
p_i					
q_i					

let $e[1..n+1, 0..n]$, $w[1..n+1, 0..n]$, and $root[1..n, 1..n]$ be new tables.
for $i = 1$ to $n+1$
    $e[i, i-1] = q_{i-1}$
    $w[i, i-1] = q_{i-1}$
for $l = 1$ to n
    for $i = 1$ to $n - l + 1$
        $j = i + l - 1$
        $e[i, j] = \infty$
        $w[i, j] = w[, i, j-1] + p_j + q_j$
        for $r = i$ to j
            $t = e[i, r-1] + e[r+1, j] + w[i, j]$
            if $i < e[i, j]$
                $e[i, j] = t$
                $root[i, j] = r$
return e and $root$

2.3.5 S17 #L4, Similar to above

(Same as Fall 2016 #L4 and S15 #L4)

Given a set of 4 keys, with the following probabilities, determine the cost and the structure of an optimal BST (binary search tree), following the tabular, bottom-up method realized in the

procedure of `OPTIMAL-BST` below to construct and fill $e[1..5, 0..4], w[1..5, 0..4]$ and $root[1..4, 1..4]$.

i	0	1	2	3	4
p_i		0.10	0.08	0.22	0.21
q_i	0.06	0.12	0.07	0.05	0.09

Construct and fill the three tables, and show the optimal BST obtained.

```
OPTIMAL-BST(p,q,n)
 1  let e[1..n+1,0..n], w[1..n+1,0..n], and root[1..n,1..n] be new tables.
 2  for i = 1 to n+1
 3          e[i,i-1] = q_{i-1}
 4          w[i,i-1] = q_{i-1}
 5  for l = 1 to n
 6          for i = 1 to n-l+1
 7                  j = i+l-1
 8                  e[i,j] = ∞
 9                  w[i,j] = w[i,j-1] + p_j + q_j
10                  for r = i to j
11                          t = e[i,r-1] + e[r+1,j] + w[i,j]
12                          if t < e[i,j]
13                                  e[i,j] = t
14                                  root[i,j] = r
15  return e and root
```

Solution

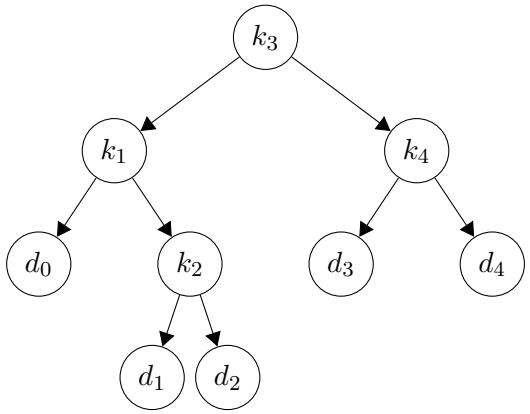
e	0	1	2	3	4
1	0.06	0.46	0.95	1.62	2.44
2	0.00	0.12	0.46	1.05	1.79
3	0.00	0.00	0.07	0.46	1.19
4	0.00	0.00	0.00	0.05	0.49
5	0.00	0.00	0.00	0.00	0.09

w	0	1	2	3	4
1	0.06	0.28	0.43	0.70	1.00
2	0.00	0.12	0.27	0.54	0.84
3	0.00	0.00	0.07	0.34	0.64
4	0.00	0.00	0.00	0.05	0.35
5	0.00	0.00	0.00	0.00	0.09

root	1	2	3	4
1	1	1	2	3
2	0	2	3	3
3	0	0	3	4
4	0	0	0	4

Here's how to use `root` to find the structure of the table.

The element $root[1,4] = 3$ gives the root node for the optimal binary search tree containing nodes k_1, k_2, k_3, and k_4. So k_3 is at the root, which means k_4 is its right child. The left subtree of k_3 contains k_1 and k_2. The remaining question is how to structure that subtree. The value of $root[1,2] = 1$ tells us that the subtree containing k_1 and k_2 has k_1 as its root. Then fill in the d's.



$1 \times (p_3) = 1 \times (0.22) = 0.22$

$2 \times (p_1 + p_4) = 2 \times (0.10 + 0.21) = 0.62$

$3 \times (q_0 + p_2 + q_3 + q_4)$
$= 3 \times (0.06 + 0.08 + 0.05 + 0.09) = 0.84$

$4 \times (q_1 + q + 2) = 4 \times (0.12 + 0.07) = 0.76$

$0.22 + 0.62 + 0.84 + 0.76 = 2.44 = e[1,4]$

F18 #L1

Describe how dynamic programming is used to construct an optimal binary search tree.

Use the following probability of p and q, obtain the expected cost of searching an optimal binary search tree constructed by dynamic programming.

i	0	1	2	3	4	5	6	7
p_i		0.04	0.06	0.08	0.06	0.10	0.12	0.10
q_i	0.06	0.06	0.06	0.06	0.05	0.05	0.05	0.05

[Editor's Note] This is exercise 15.5-2, except that they changed two of the numbers. It's different from other similar questions in the comps in that they didn't give us the code to follow.

Why did they choose such a bloody long and tedious problem, where one calculation error can propagate through the entire problem? They're just making more work for themselves grading it. This is an $O(n^3)$ problem. A problem with $n = 4$ would have told them whether we knew how to

do it.

Solution to the First Part

Four steps of developing a dynamic-programming algorithm.

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution.
4. Construct an optimal solution from computed information.

Solution to the Second Part

e	0	1	2	3	4	5	6	7
1	0.06	0.28	0.62	1.02	1.43	1.95	2.56	3.14
2		0.06	0.30	0.68	1.01	1.53	2.08	2.63
3			0.06	0.32	0.65	1.08	1.60	2.15
4				0.06	0.28	0.65	1.09	1.59
5					0.05	0.30	0.72	1.12
6						0.05	0.32	0.72
7							0.05	0.30
8								0.05

w	0	1	2	3	4	5	6	7
1	0.06	0.16	0.28	0.42	0.53	0.68	0.85	1.00
2		0.06	0.18	0.32	0.43	0.58	0.75	0.90
3			0.06	0.20	0.31	0.46	0.63	0.78
4				0.06	0.17	0.32	0.49	0.64
5					0.05	0.20	0.37	0.52
6						0.05	0.22	0.37
7							0.05	0.20
8								0.05

root	1	2	3	4	5	6	7
1	1	2	2	3	3	3	4
2		2	3	3	3	5	5
3			3	3	4	5	5
4				4	5	5	6
5					5	6	6
6						6	6
7							7

Here's how to use `root` to build the tree.

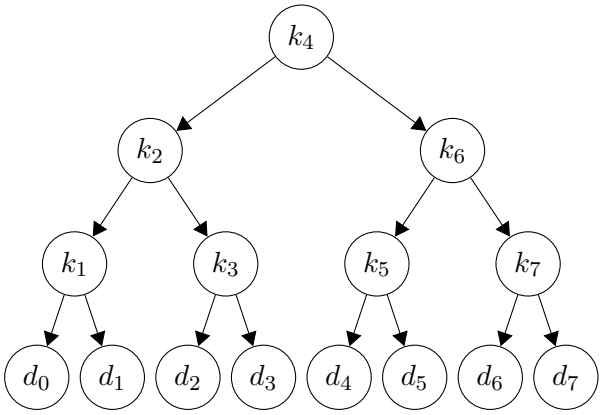
The value $root[1,7] = 4$ indicates that k_4 is the root of the subtree containing $k_1, \ldots, k_7$, *i.e.* the entire tree.

The left subtree contains k_1, k_2, and k_3, and the right subtree contains k_5, k_6 and k_7.

The root of the subtree containing k_1, k_2, and k_3 is k_2 because $root[1,3] = 2$. Its left child is the subtree containing k_1, and its right child is the subtree containing k_3.

The root of the subtree containing k_5, k_6, and k_7 is k_6 because $root[5,7] = 6$. Its left child is the subtree containing k_5, and its right child is the subtree containing k_7.

Add the dummy nodes.



$1 \times (p_4) = 1 \times 0.06 = 0.06$

$2 \times (p_2 + p_6) = 2 \times (0.06 + 0.12) = 0.36$

$3 \times (p_1 + p_3 + p_5 + p_7)$
$= 3 \times (0.04 + 0.08 + 0.10 + 0.10) = 0.96$

$4 \times (q_0 + q_1 + q_2 + q_3 + q_4 + q_5 + q_6 + q_7)$
$= 4 \times (0.06 + 0.06 + 0.06 + 0.06 + 0.05 + 0.05 + 0.05 + 0.05) = 17.6$

$0.06 + 0.36 + 0.96 + 1.76 = 3.14 = e[1,7]$

2.4 Height Balanced Binary Search Tree (AVL Tree), §13, Problem 13-3

An AVL Tree is a binary search tree that is **height balanced**; that is, for each node, x, the heights of the left and right subtrees differ by at most 1.

2.4.1 Old Exam Questions

S17 #S1

a. Define height balanced binary tree.
b. Write a pseudo code to determine whether a tree is height balanced?
c. Obtain the tight bound of your algorithm.

S18 #S5, Same as above

a. Define height balanced binary tree
b. Write a pseudo code to determine whether a tree is height balanced?
c. Obtain a tight bound of your algorithm.

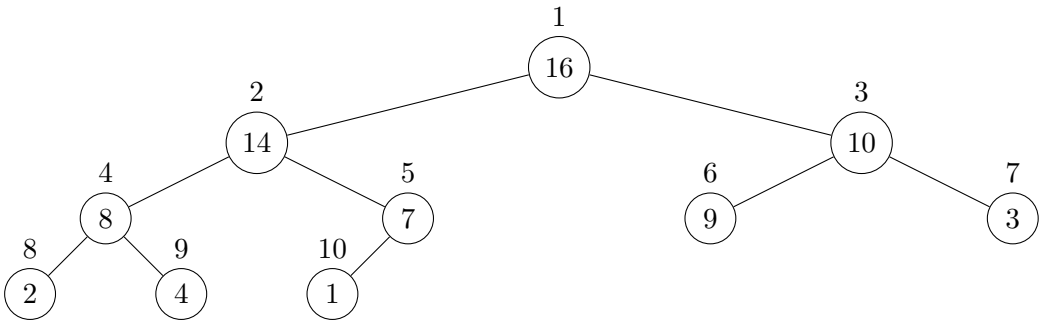
2.5 Heaps

Heap Nearly complete binary tree. All levels, except perhaps the lowest, are complete, and the bottom row is filled from the left.

Max-heap property $A[Parent(i)] \geq A[i]$ The children are no larger than their parent.

Min-heap property $A[Parent(i)] \leq A[i]$ The children are no less than their parent.

Max-heap $\{16, 14, 10, 8, 7, 9, 3, 2, 4, 1\}$ as a binary tree.



Building a max heap takes $O(n)$ time.

If you have built a max heap and swap out the root node, then it takes $O(\lg n)$ steps to restore the heap.

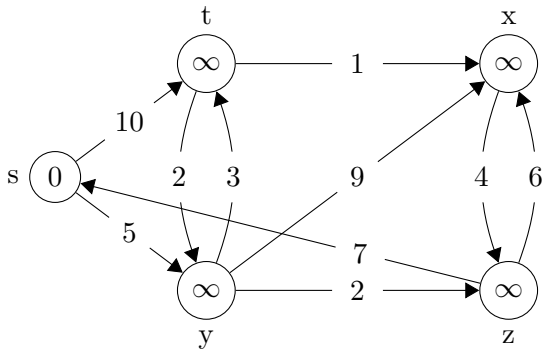
Since the head node of a max heap is always the largest node, a method that works to sort the heap is to take out the head node, replace it with the last node, restore the heap, and repeat until the heap is empty. Since this sort takes $n-1$ steps, and each step takes $O(\lg n)$ time, the sort takes $O(n \lg n)$ time.

2.5.1 Old Exam Questions

Spring 2019 #L4

The Dijkstra's algorithm (DIJ) solves the single-source shortest-path problem in a weighted directed graph $G = (V, E)$. Given the graph G below, follow DIJ to find shortest paths from vertex s to all other vertices, with all predecessor edges shaded and estimated distance values from s to all vertices provided at the end of each iteration.

What is the time complexity of DIJ for a general graph $G = (V, E)$, if the candidate vertices are kept in a binary min-heap?



Solution to second part

The time complexity for Dijkstra's algorithm has two parts. We run `Extract-Min` once per vertex, and `Relax` once for each edge, so the total time is

$$V \cdot (\text{Time for Extract-Min}) + E \cdot (\text{Time for Relax})$$

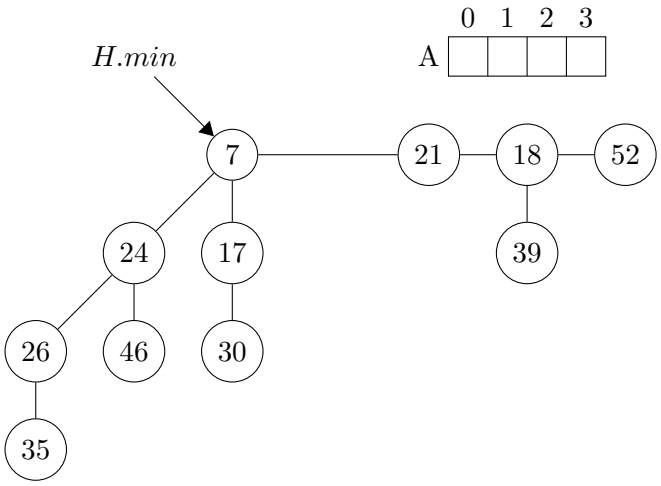
The time for each relaxation is constant. The time for the `Extract-Min` is the time to search for Q the vertex with the least distance from the source.

If we store the vertex distances in an array, then the cost of finding the one with least distance is $O(V)$, so the total cost of Dijkstra is $O(V^2 + E)$, which is $O(V^2)$ because $E \leq V^2$.

If, on the other hand, we store the vertices in a min-priority queue with a binary min-heap, searching for the min is faster, but finding a vertex to delete is slower. Both now take $O(\lg V)$ time. Each `Extract-Min` is $O(\lg V)$, and each `Relax` takes $O(\lg V)$, so the total time is now $O(V \lg V + E \lg V) = O((V + E) \lg V) = O(E \lg V)$. This method is faster if $E \lg V < V^2$, which is true if the graph is sparse.

Spring 2018 #L2

A Fibonacci min-heap relies on the procedure of CONSOLIDATE to merge min-heaps in the root list upon the operation of extracting the minimum node. Given the following Fibonacci min-heap, show every consolidation step and the final heap result after $H.min$ is extracted, with the aid of A.



Spring 2017 #S5

a. What are the properties of min heap and max heaps.
b. What is the preferred data structure of implementing binary heap, also justify your answer.
c. What is the time complexity of merging two different min heaps each of size n and m.

Solutions

a. The min-heap property is that, in a min-heap, every node i other than the root has the property $A[parent] \leq A[i]$.
   The max-heap property is that, in a max-heap, every node i other than the root has the property $A[parent] \geq A[i]$.
b. The preferred data structure for implementing a binary heap is the *priority queue*. It extends max-heap and min-heap to allow insertion, and changing (increasing for max-heap, decreasing for min-heap) the value of a key.
   [I would have answered this question with Fibonacci Heap, but that's not binary. Also, while it's of theoretical interest and faster in a few applications, in most applications it is not.]
c. The time complexity of merging two Fibonacci min heaps is $O(1)$ because it's not a tree, and you can just stick them together at the root and find the min of the root list.
   To merge two binary heaps, you can concatenate the two arrays to make a binary heap of size $n + m$, possibly in constant time, at worst in linear time, and then run `Min-Heapify`, which partially sorts it so that it has the min-heap property, and will run in $O(\lg(n + m))$ time. Therefore, the time complexity of merging two different min heaps of size n and m is $O(\lg(n + m))$.

Fall 2016 #S4

Same as above.

1. What are the properties of min heap and max heaps.
2. What is the preferred data structure of implementing binary heap, also justify your answer.
3. What is the time complexity of merging two different min heap with sizes of n and m.

Fall 2015 #S7

Mark True or False against the following statements.

a. (Same as S15 #9a) A binary search tree of size N will always find a key in at most $O(\log N)$ time.
b. An optimal binary search is not necessary a balanced tree.
c. A binary heap always maintains a balanced tree as practical as it can be.
d. To implement a priority queue binomial heap is preferred over binary heap.
e. A graph formed by strongly connected components, a strongly connected components graph (SCC) is always a minimum spanning tree.

Solutions

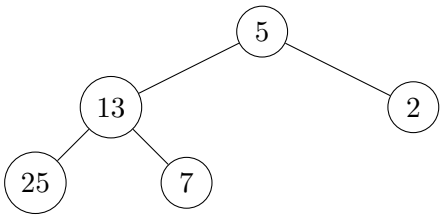
a. False. A balanced binary search tree will always find a key in at most $O(\lg N)$ time, but a binary search tree does not have to be, and often is not, balanced. The worst-case scenario is that the tree only has a single branch, and the search time is $O(N)$.
b. True.
c. True.
d. True.
e. False. It will be a spanning tree, but not necessarily minimal.

Spring 2015 #S5

Use a binary tree representation to illustrate every operation of `MIN HEAPSORT` involved when sorting array $A = \{5, 13, 2, 25, 7\}$ without auxiliary storage.

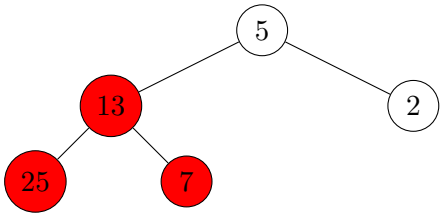
[Note: The `Max Heapsort` version of this question is the example on page 161.]

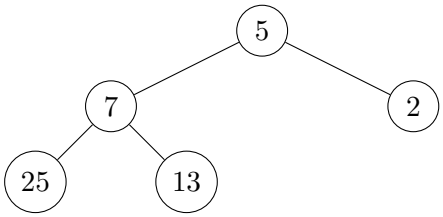
First make the heap.



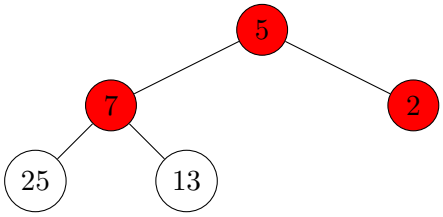
Then `Min-Heapify` it.

Find the min of these three, and swap it with the parent.

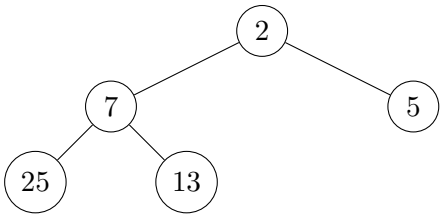




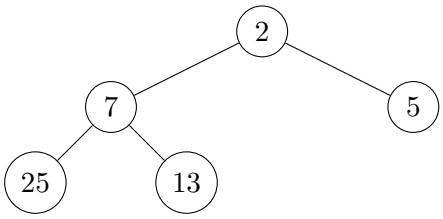
Find the min of these three, and swap it with the parent.



Now we have a min heap.

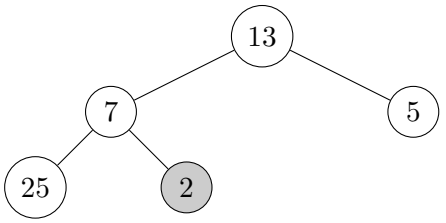


At each step, swap the root with the last leaf, decrement the number of elements in the heap (but not the array), and restore the min-heap using `Min-Heapify`. Repeat until the heap is empty.



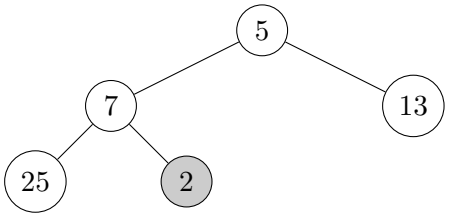
$A = [2, 7, 5, 25, 13],\ N = 5$

Swap



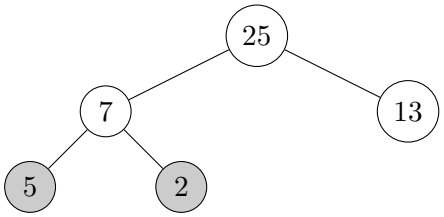
$A = [13, 7, 5, 25, 2],\ N = 4$

Heapify



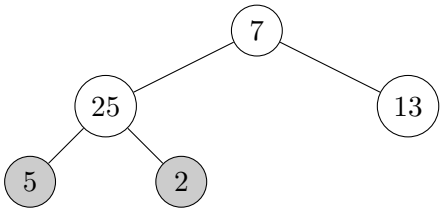
$A = [5, 7, 13, 25, 2],\ N = 4$

Swap



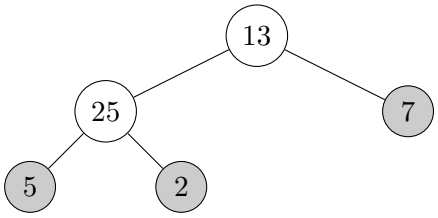
$A = [25, 7, 13, 5, 2],\ N = 3$

Heapify



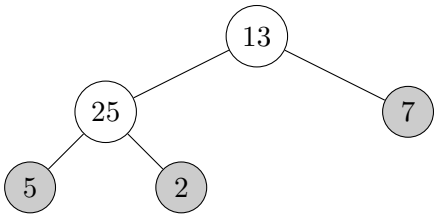
$A = [7, 25, 13, 5, 2],\ N = 3$

Swap



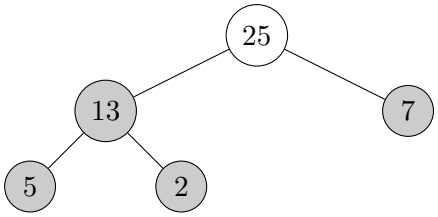
$A = [13, 25, 7, 5, 2],\ N = 2$

Heapify (no change)



$A = [13, 25, 7, 5, 2],\ N = 2$

Swap



$A = [25, 13, 7, 5, 2],\ N = 1$

Chapter 3

Sorting

3.1 Insertion Sort

3.1.1 Algorithm

```
def Insertion_Sort(A):
    for j in range (1, len(A)):
        key = A[j]
        i = j-1
        print (A[:j], key)
        while i>-1 and A[i]>key:
            A[i+1] = A[i]
            print ("    ", A[:j+1])
            i = i-1
        A[i+1] = key
        print ("    ", A[:j+1])
    return A

A = [8,7,6,5,4,3,2,1]
Insertion_Sort(A)
```

3.1.2 Time Complexity

In the worst case, the `while` loop has to go the way to the bottom for every value of j. This case happens when the array is sorted in decreasing order. In that case, the recurrence is

$$T(n) = T(n-1) + cn = \Theta(n^2)$$

In the best case, for every value of j, $A[i] = A[j-1] \leq key$, so we never engage the `while` loop. This happens when the array is already sorted in increasing order, so the time complexity is linear, $\Theta(n)$.

3.2 Quicksort

3.2.1 Code and Example

```
count = 0

def Partition(A, p, r):
    x = A[r]
    i = p-1
    for j in range (p, r):
        if A[j]<=x:
            i = i+1
            A[i], A[j] = A[j], A[i]
    A[i+1], A[r] = A[r], A[i+1]
    print ("  x, A = ", x, A)
    return i+1

def Quicksort(A, p, r):
    print ("\n(p,r) = ",p, r)
    print ("Partition the array ", A[p:r+1])
    print ("Pivot at A[r] =", A[r])
    if p<r:
        q = Partition(A, p, r)
        global count
        count = count + 1
        print ("    dividing the array into subarrays", A[p:q], ",", A[q], ", and",
        Quicksort(A, p, q-1)
        Quicksort(A, q+1, r)


#A = [2,8,7,1,3,5,6,4]
#A = [1,2,3,4,5,6,7,8]
A = [1,3,4,5,7,6,4]
print ("A = ", A)
print ()

#Partition(A,0,len(A)-1)
Quicksort(A, 0, len(A)-1)
print ("Number of calls of Quicksort function = ", count)
```

```
A =  [2, 8, 7, 1, 3, 5, 6, 4]
```

```
(p,r) =  0 7
Partition the array  [2, 8, 7, 1, 3, 5, 6, 4]
Pivot at A[r] = 4
  x, A =  4 [2, 1, 3, 4, 7, 5, 6, 8]
    dividing the array into subarrays [2, 1, 3] , 4 , and [7, 5, 6, 8] .

(p,r) =  0 2
Partition the array  [2, 1, 3]
Pivot at A[r] = 3
  x, A =  3 [2, 1, 3, 4, 7, 5, 6, 8]
    dividing the array into subarrays [2, 1] , 3 , and [] .

(p,r) =  0 1
Partition the array  [2, 1]
Pivot at A[r] = 1
  x, A =  1 [1, 2, 3, 4, 7, 5, 6, 8]
    dividing the array into subarrays [] , 1 , and [2] .

(p,r) =  0 -1
Partition the array  []
Pivot at A[r] = 8

(p,r) =  1 1
Partition the array  [2]
Pivot at A[r] = 2

(p,r) =  3 2
Partition the array  []
Pivot at A[r] = 3

(p,r) =  4 7
Partition the array  [7, 5, 6, 8]
Pivot at A[r] = 8
  x, A =  8 [1, 2, 3, 4, 7, 5, 6, 8]
    dividing the array into subarrays [7, 5, 6] , 8 , and [] .

(p,r) =  4 6
Partition the array  [7, 5, 6]
```

```
Pivot at A[r] = 6
  x, A =  6 [1, 2, 3, 4, 5, 6, 7, 8]
    dividing the array into subarrays [5] , 6 , and [7] .

(p,r) =  4 4
Partition the array  [5]
Pivot at A[r] = 5

(p,r) =  6 6
Partition the array  [7]
Pivot at A[r] = 7

(p,r) =  8 7
Partition the array  []
Pivot at A[r] = 8
```

3.2.2 Time Complexity

The algorithm recursively divides the array in two pieces, so it looks at $\Theta(\lg n)$ subarrays. On each subarray, it reads through left to right, so the partitioning on the subarray of length $k < n$ is $\Theta(k)$.

Worst-Case Partitioning

The worst-case partitoning happens, ironically, in an array that's already sorted in increasing order. The partitioning routine produces one subproblem with $n - 1$ elements and one problem with zero elements. If it happens in every recursive call, the recurrence for the running time is

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \end{aligned}$$

which evaluates to $\Theta(n^2)$.

Best-Case Partitioning

Each partitioning splits the array of length n into subarrays of lengths $n/2$ and $n/2 - 1$, with recurrence $T(n) = 2T(n/2) + \Theta(n)$ that evaluates to $\Theta(n \lg n)$.

Average-Case Partitioning

Much closer to best case than worst case.

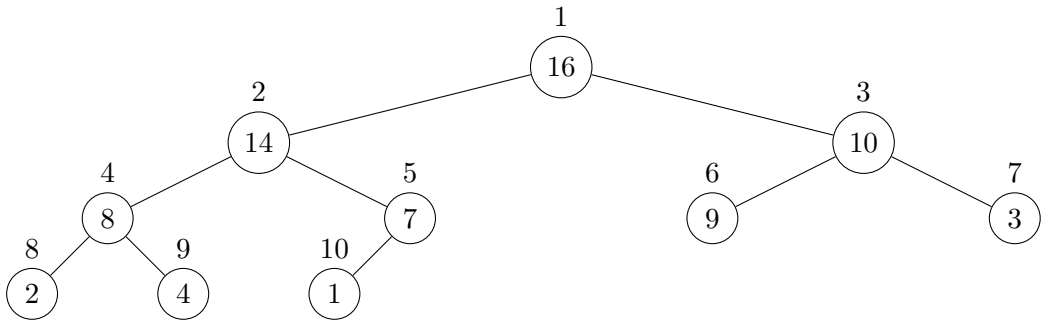
3.3 Heap Sort

Heap Nearly complete binary tree. All levels, except perhaps the lowest, are complete, and the bottom row is filled from the left.

Max-heap property $A[Parent(i)] \geq A[i]$ The children are no larger than their parent.

Min-heap property $A[Parent(i)] \leq A[i]$ The children are no less than their parent.

Max-heap $\{16, 14, 10, 8, 7, 9, 3, 2, 4, 1\}$ as a binary tree.



Building a max heap takes $O(n)$ time.

If you have built a max heap and swap out the root node, then it takes $O(\lg n)$ steps to restore the heap.

Since the head node of a max heap is always the largest node, a method that works to sort the heap is to take out the head node, replace it with the last node, restore the heap, and repeat until the heap is empty. Since this sort takes $n-1$ steps, and each step takes $O(\lg n)$ time, the sort takes $O(n \lg n)$ time.

3.4 Old Exam Questions: Sorting

Fall 2018 #S2

a. What is the lower bound for comparisons based sorting algorithm? (Outline the justification of your answer.)
b. What is the strategy behind greedy algorithm?

Fall 2016 #S2

1. Briefly describe a quick sort algorithm for sorting objects in an ascending order of their keys.
2. What is the best and worst case time complexity of quick sort and the reason for such complexity?

Solution

1. (a) Take the last element of the array, $x = A[n]$.
   (b) Partition the remaining array into two subarrays, B whose elements are no more than x, and C whose elements are greater than x, rearranging A so that it is $B + \{x\} + C$ (with "+" meaning concatenation).
   (c) Recurse on both B and C.
2. The best time complexity happens when each partition divides A "evenly" into subarrays of length $n/2$ and $n/2 - 1$. The recurrence is then $T(n) = 2T(n/2) + cn = \Theta(n \lg n)$.
   The worst case happens the last element in each subarray is the largest value in the subarray, meaning that the partition is into subarrays of length $n - 1$ and 0, giving a recurrence $T(n) = T(n-1) + cn = \Theta(n^2)$.
   The worst-time case happens, ironically, when the original array is already sorted in ascending order. The average time complexity is much closer to the best-time than the worst-time.

Fall 2015 #S3

What is the time complexity of insertion sort algorithm? Suppose you already know the total number of keys and range of the key values. What would be the best and worst results you will get when sorting n items? In addition to the previous information, if you already know that the key values are uniformly distributed, what would be your best and worst results? (Make sure you sketch the algorithm and provide rationale of your expected results. Do NOT derive the result.)

Solution

See above about best and worst cases.

I don't know that being uniformly distributed makes a difference. ?????

Spring 2015 #S2

In terms of run time efficiency, compare and contrast quick sort and merge sort. What is the best and the worst case time complexity of the quick sort algorithm? Also state under what conditions one may expect these two extreme cases.

Solution

Merge sort and quick sort both use a divide-and-conquer strategy.

Merge sort divides the array of length n into n subarrays of length 1, then recombines them two at a time into sorted arrays. The merge of two subarrays of length n takes $\Theta(n)$ time. The recurrence is $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \lg n)$.

Quick sort partitions the array of length n into two subarrays of length m and $n - m - 1$. In the best case, $m = n/2$, and in the worst case, $m = 0$. In the best case, the recurrence is

$T(n) = 2T(n/2) + cn = \Theta(n \lg n)$, and in the worst, $T(n) = T(n-1) + cn = \Theta(n^2)$. The worst case happens, ironically, when the array is already sorted.

Chapter 4

NP-Complete

4.1 NP-Completeness §34

P Class of problems *solvable* in polynomial time, *i.e.* $O(n^k)$ for some constant k.

NP Class of problems *verifiable* in polynomial time.
Non-deterministic Polynomial time or *Non-deterministic Polynomial acceptance problems*

NP-Complete Class of problems in NP that is as hard as any other problem in NP.

Optimization Problem Given an undirected graph G and vertices u and v, find a path from u to v that uses the fewest edges.

Decision Problem Given an undirected graph G, vertices u and v, and a constant k, is there a path from u to v consisting of at most k edges?

Instance An instance of a problem is a particular set of inputs to the problem.

NP deals with decision problems, but the decision problem is no harder than the corresponding optimization problem. If we show that the decision problem is NP-complete, then the optimization problem is NP-complete.

4.1.1 Polynomial-time Reduction Algorithm

Given a decision problem A we want to show can be solved in polynomial time, if we can show it is reducible in polynomial time to a decision problem B known to be solvable in polynomial time, then we know A can be solved in polynomial time.

1. Given an instance α of problem A, use a polynomial-time reduction algorithm to reduce it to an instance β of problem B.
2. Run the polynomial-time decision algorithm for B on the instance β.
3. Use the answer for β as the answer for α.

If the answer for each β is the same as the answer for the corresponding α, and if the transformation from α to β takes polynomial time, and the solution for β takes polynomial time, then we

can use the transformation to β to solve α in polynomial time.

4.1.2 Showing a Problem is NP-Complete

We have a problem B that we want to show is not in P.

Assume that we have a decision problem A for which we know that no polynomial-time algorithm can exist. [Note that we are assuming that P is a proper subset of NP.]

Also assume that we have a polynomial-time transformation that maps instances of A to instances of B.

Prove by contradiction that $B \notin P$.

Suppose B has a polynomial-time algorithm. Then, since we have a polynomial-time reduction transforming instances of A to instances of B, there is a way to solve A in polynomial time; however, this conclusion contradicts the given information that $A \notin P$. Therefore, $B \notin P$.

4.1.3 Circuit Satisfiability §34.4

To show that a problem, B, is NP-Complete, we first have to have a problem, A, known to be NP-Complete, so that we can find a polynomial-time reduction transforming instances of A into instances of B.

Our first NP-Complete problem will be the *Boolean satisfiability problem*, a.k.a. *propositional satisfiability*, *SAT*, *k-conjunctive normal form satisfiability*

A *truth assignment* for a boolean combinational circuit is a set of boolean input values. A one-output boolean combinational circuit is *satisfiable* if it has a *satisfying assignment*, that is, a truth assignment that causes the output of the circuit to be 1. The *circuit satisfiability problem* is, "Given a boolean combinational circuit composed of AND, OR, and NOT gates, is it satisfiable?"

In general, a *boolean combinational element* is a well-defined function that takes in a constant number of boolean inputs and returns a constant number of boolean outputs. In this problem, the boolean combinational elements are the logic gates (AND, OR, NOT), each of which has a constant number of boolean inputs and one boolean output.

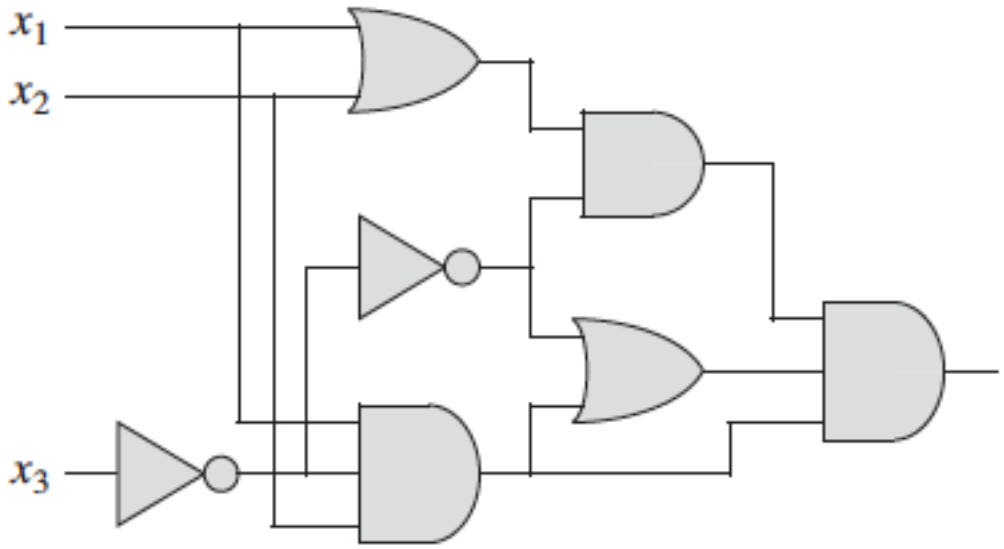
A *boolean combinational circuit* has one or more boolean combinational elements (logic gates) connected by *wires*. Each wire connects the output of one gate to the input of (at least one) other, making the output of the first gate an input for the second. Wires can also take inputs from elsewhere and send outputs elsewhere.

The *size* of a boolean combinational circuit is the number of boolean combinational elements (gates) plus the number of wires (including input/output from/to elsewhere). If k is the number of inputs in circuit C, then the size of the circuit is polynomial in k, and checking the problem has lower bound $\Omega(2^k)$ time. The time is superpolynomial in the size of the circuit.

`https://en.wikipedia.org/wiki/Boolean_satisfiability_problem`

Example

Figure 34.8, page 1072



I have labeled each of the logic gates, going left to right and top to bottom.

$a = \neg x_3$
$b = x_1 \vee x_2$
$c = \neg a = x_3$
$d = x_1 \wedge a \wedge x_2 = x_1 \wedge x_2 \wedge \neg x_3$
$e = b \wedge c = (x_1 \vee x_2) \wedge x_3$
$f = c \vee d = x_3 \vee (x_1 \wedge x_2 \wedge \neg x_3) = (x_1 \wedge x_2) \vee x_3$
$g = e \wedge f \wedge d = ((x_1 \vee x_2) \wedge x_3) \wedge ((x_1 \wedge x_2) \vee x_3) \wedge (x_1 \wedge x_2 \wedge \neg x_3) = (x_1 \wedge x_2) \wedge (x_3 \wedge \neg x_3)$

To test all of the inputs, we have 2^n inputs, where n is the number of boolean variables. Then we run each of those inputs through k tests, where k is the number of logic gates. So we have $k \cdot 2^n$ tests, meaning that testing all of the inputs is $O(2^n)$.

			a	b	c	d	e	f	g
x_1	x_2	x_3	$\neg x_3$	$x_1 \vee x_2$	$\neg a$	$x_1 \wedge a \wedge x_2$	$b \wedge c$	$c \vee d$	$d \wedge e \wedge f$
0	0	0	1	0	0	0	0	0	0
0	0	1	0	0	1	0	0	1	0
0	1	0	1	1	0	0	0	0	0
0	1	1	0	1	1	0	1	1	0
1	0	0	1	1	0	0	0	0	0
1	0	1	0	1	1	0	1	1	0
1	1	0	1	1	0	1	0	1	0
1	1	1	0	1	1	0	1	1	0

Direct method: The problem is not satisfiable because we tested each of the instances and all of them failed.

Heuristic (?) method: The problem is not satisfiable because $d \wedge e = (x_1 \wedge x_2 \wedge \neg x_3) \wedge ((x_1 \vee x_2) \wedge x_3) = x_1 \wedge (x_1 \vee x_2) \wedge x_2 \wedge x_3 \wedge \neg x_3$, which depends on $x_3 \wedge \neg x_3$ being true, which it is not for either value of x_3.

Lemma. *The circuit-satisfiability problem belongs to NP.*

Proof. To "belong to NP" means to be verifiable in polynomial time. We will give a sketch of a polynomial-time algorithm to verify an instance of the problem.

Two-input polynomial-time algorithm, A.

One input is a boolean combinational circuit, C

Second input is a *certificate* corresponding to an assignment of boolean values to the wires in the circuit C.

For each logic gate in C, the algorithm A checks that the values of the output wire(s) are correctly computed from the values of the input wire(s). If all of the computations are correct and the output of the entire circuit is 1, then the algorithm outputs 1. Otherwise, (if any of the computations are incorrect, or if the output of the entire circuit is 0), then the algorithm A returns 0.

For a satisfiable circuit C, there exists a certificate whose length is polynomial in the size of C and that causes A to output 1. [Actually, the length of the certificate doesn't need to be larger than C.] For an unsatisfiable circuit C, no certificate will return 1.

Since A runs in (no worse than) polynomial time, we can verify the circuit satisfiability problem, so it belongs to NP. □

Lemma. *The circuit-satisfiability problem is NP-hard.*

Proof. (To Do?) □

Theorem. *The circuit-satisfiability problem is NP-complete.*

Proof. Since the problem is both NP and NP-hard, by definition it is therefore NP-complete. □

3-CNF or 3-P SAT

A *literal* is an occurrence of a variable or its negation. A boolean formula is in *conjunctive normal form* if it is expressed as an AND of clauses, each of which is an OR of one or more literals. A boolean formula is in *3-conjunctive normal form* or *3-CNF* if each clause has exactly three distinct literals.

The 3-CNF-SAT problem asks whether a given boolean formula ϕ in 3-CNF is satisfiable.

Theorem. *Satisfiability of boolean formulas in 3-conjunctive normal form is NP-complete.*

Sketch of proof. Need to prove that 3-CNF-SAT $\in$ NP and 3-CNF-SAT $\in$ NP-hard.

To show that 3-CNF-SAT $\in$ NP, we show that there is a polynomial-time algorithm, A, to verify an instance of the problem. The algorithm A has two inputs, the boolean combinational circuit C

and a certificate corresponding to an assignment of boolean values to the wires in C. To verify, we need to check that, for each gate, the boolean values on the input wires give the result on the output wire, and that the value of the final output wire is 1. This verification algorithm is $O(|C|)$, where $|C|$ is the number of wires plus the number of gates; therefore, we have a polynomial-time algorithm that verifies any instance of the problem; therefore, 3-CNF-SAT $\in$ NP.

To show that 3-CNF-SAT $\in$ NP-hard, we will show that SAT $\leq_p$ 3-CNF-SAT, where "$\leq_p$" is the *polynomial-time reducibility relation.* We do that by showing that every boolean combinational circuit satisfactiability problem can be transformed in polynomial time into a 3-conjunctive normal form satisfiability problem. A general SAT problem cannot take longer than 3-CNF-SAT problems, because if it did, you could transform it into a 3-CNF-SAT problem and solve it; therefore, SAT $\leq_p$ 3-CNF-SAT, and since SAT $\in$ NP-Complete, we have 3-CNF-SAT $\in$ NP-hard.

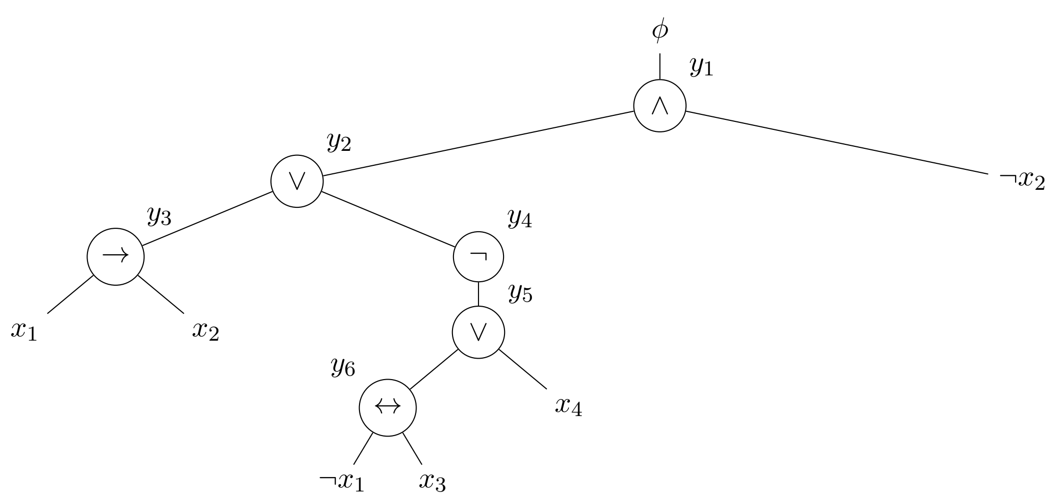
Since 3-CNF-SAT $\in$ NP and 3-CNF-SAT $\in$ NP-hard, 3-CNF-SAT $\in$ NP-Complete. *Q.E.D.*

Example on page 1082 of reduction algorithm from SAT to 3-CNF-SAT

$$\phi = ((x_1 \to x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

$$\phi = \underbrace{\underbrace{(\underbrace{(x_1 \to x_2)}_{y_3=x_1\to x_2} \vee \underbrace{\neg(\underbrace{(\underbrace{(\neg x_1 \leftrightarrow x_3)}_{y_6=\neg x_1\leftrightarrow x_3} \vee x_4)}_{y_5=y_6\vee x_4})}_{y_4=\neg y_5})}_{y_2=y_3\vee y_4} \wedge \neg x_2}_{y_1=y_2\wedge\neg x_2}$$

Binary Parse Tree



				y_3	y_6	y_5	y_4	y_2	y_1
x_1	x_2	x_3	x_4	$x_1 \to x_2$	$\neg x_1 \leftrightarrow x_3$	$y_6 \vee x_4$	$\neg y_5$	$y_3 \vee y_4$	$y_2 \wedge \neg x_2$
0	0	0	0	1	0	0	1	1	1
0	0	0	1	1	0	1	0	1	1
0	0	1	0	1	1	1	0	1	1
0	0	1	1	1	1	1	0	1	1
0	1	0	0	1	0	0	1	1	0
0	1	0	1	1	0	1	0	1	0
0	1	1	0	1	1	1	0	1	0
0	1	1	1	1	1	1	0	1	0
1	0	0	0	0	1	1	0	0	0
1	0	0	1	0	1	1	0	0	0
1	0	1	0	0	0	0	1	1	1
1	0	1	1	0	0	1	0	0	0
1	1	0	0	1	1	1	0	1	0
1	1	0	1	1	1	1	0	1	0
1	1	1	0	1	0	0	1	1	0
1	1	1	1	1	0	1	0	1	0

See `Reduction_1082.py`

We can rewrite a logical function f on a and b, $f(a, b)$, by introducing a variable y that has the same value as $f(a, b)$, *i.e.* y is true when $f(a, b)$ is true and y is false when $f(a, b)$ is false. They have the same value when $y \leftrightarrow f(a, b)$ is true. We know that $f(a, b)$ is true when $y \wedge (y \leftrightarrow f(a, b))$ is true. We can chain these together to get a formula ϕ' equivalent to ϕ, as a conjunction of clauses ϕ'_i with at most three literals.

$$\begin{aligned}\phi' = y_1 &\wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\ &\wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \\ &\wedge (y_3 \leftrightarrow (x_1 \to x_2)) \\ &\wedge (y_4 \leftrightarrow (\neg y_5)) \\ &\wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \\ &\wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))\end{aligned}$$

Below are the rows of the truth table for which $\phi' = 1$. Note that they are the same values for x_i as in ϕ.

x_1	x_2	x_3	x_4	y_1	y_2	y_3	y_4	y_5	y_6	ϕ'
0	0	0	0	1	1	1	1	0	0	1
0	0	0	1	1	1	1	0	1	0	1
0	0	1	0	1	1	1	0	1	1	1
0	0	1	1	1	1	1	0	1	1	1
1	0	1	0	1	1	0	1	0	0	1

Now we have a formula ϕ' that is a conjunction of clauses ϕ'_i, each of which has at least three literals. To be in 3-conjunctive normal form, we need it to be a conjunction of clauses, each of which is a disjunction of exactly three literals. To go from ϕ' to ϕ'', follow these steps.

1. Make a truth table for each clause ϕ'_i.
2. Using each entry that evaluates to 0, build a formula in disjunctive normal for (an OR of ANDs) that is equivalent to $\neg\phi'_i$.
3. Negate this formula and convert it into a CNF formula ϕ''_i using DeMorgan's laws for propositional logic, particularly that $\neg(a \wedge b) = \neg a \wedge \neg b$ to complement all literals, changing ORs to ANDs and vice versa. Now we have converted each clause ϕ'_i into a conjunctive normal form ϕ''_i with at most three literals.
4. If a clause ϕ''_i has fewer than three literals, pad it.
5. Take the conjunction of all of the clauses.

x_2	y_1	y_2	$y_2 \wedge \neg x_2$	$y_1 \leftrightarrow (y_2 \wedge \neg x_2)$	DNF for $\neg\phi'_2$	CNF for ϕ''_2
1	1	1	0	0	$x_2 \wedge y_1 \wedge y_2$	$\neg x_2 \vee \neg y_1 \vee \neg y_2$
1	1	0	0	0	$x_2 \wedge y_1 \wedge \neg y_2$	$\neg x_2 \vee \neg y_1 \vee y_2$
1	0	1	0	1		
1	0	0	0	1		
0	1	1	1	1		
0	1	0	0	0	$\neg x_2 \wedge y_1 \wedge \neg y_2$	$x_2 \vee \neg y_1 \vee y_2$
0	0	1	1	0	$\neg x_2 \wedge \neg y_1 \wedge y_2$	$x_2 \vee y_1 \vee \neg y_2$
0	0	0	0	1		

CNF: $\phi''_2 = (\neg x_2 \vee \neg y_1 \vee \neg y_2) \wedge (\neg x_2 \vee \neg y_1 \vee y_2) \wedge (x_2 \vee \neg y_1 \vee y_2) \wedge (x_2 \vee y_1 \vee \neg y_2)$

x_2	y_1	y_2	$\neg x_2 \vee \neg y_1 \vee \neg y_2$	$\neg x_2 \vee \neg y_1 \vee y_2$	$x_2 \vee \neg y_1 \vee y_2$	$x_2 \vee y_1 \vee \neg y_2$	ϕ''_2
1	1	1	0	1	1	1	0
1	1	0	1	0	1	1	0
1	0	1	1	1	1	1	1
1	0	0	1	1	1	1	1
0	1	1	1	1	1	1	1
0	1	0	1	1	0	1	0
0	0	1	1	1	1	0	0
0	0	0	1	1	1	1	1

ϕ'_5 has only two literals.

y_4	y_5	$y_4 \leftrightarrow \neg y_5$	$\neg\phi'_5$	ϕ''_5	CNF for ϕ''_5
1	1	1			
1	0	0	$y_4 \wedge \neg y_5$	$\neg y_4 \vee y_5$	$(\neg y_4 \vee y_5 \vee p) \wedge (\neg y_4 \vee y_5 \vee \neg p)$
0	1	0	$\neg y_4 \wedge y_5$	$y_4 \vee \neg y_5$	$(y_4 \vee \neg y_5 \vee p) \wedge (y_4 \vee \neg y_5 \vee \neg p)$
0	0	1			

$$\phi''_5 = (\neg y_4 \vee y_5 \vee p) \wedge (\neg y_4 \vee y_5 \vee \neg p) \wedge (y_4 \vee \neg y_5 \vee p) \wedge (y_4 \vee \neg y_5 \vee \neg p)$$

ϕ'_1 has only one literal.

y_1	$\neg\phi'_1$	ϕ''_1	$CNF for \phi''_1$
1			
0	$\neg y_1$	y_1	$(y_1 \vee p \vee q) \wedge (y_1 \vee p \vee \neg q) \wedge (y_1 \vee \neg p \vee q) \wedge (y_1 \vee \neg p \vee \neg q)$

This p, q thing might make more sense if we put it in the table.

y_1	p	q	y_1	$\neg\phi'_1$	CNF for ϕ''_1
1	1	1	1		
1	1	0	1		
1	0	1	1		
1	0	0	1		
0	1	1	0	$\neg y_1 \wedge p \wedge q$	$y_1 \vee \neg p \vee \neg q$
0	1	0	0	$\neg y_1 \wedge p \wedge \neg q$	$y_1 \vee \neg p \vee q$
0	0	1	0	$\neg y_1 \wedge \neg p \wedge q$	$y_1 \vee p \vee \neg q$
0	0	0	0	$\neg y_1 \wedge \neg p \wedge \neg q$	$y_1 \vee p \vee q$

This conversion to 3-CNF takes polynomial time, so we have a polynomial-time reduction from SAT to 3-CNF-SAT; therefore, 3-CNF-SAT is at least as hard as SAT. Since SAT is NP-complete and SAT $\leq_p$ 3-CNF-SAT, we have 3-CNF-SAT is NP-complete.

2-CNF-SAT

A 2-CNF is boolean formula in conjunctive normal form with exactly two literals per clause, like $(a \vee b) \wedge (\neg a \vee b) \wedge (a \vee c)$.

The interesting thing about 2-CNF is that $a \vee b \equiv \neg a \rightarrow b \equiv \neg b \rightarrow a$. Use the pairs of implications to make an implication graph, a directed graph that is skew symmetric (if you swap the negations

A 2-CNF formula is satisfiable if and only if there is no variable that belongs to the same strongly connected component as its negation.

To show that a boolean expression in 2-conjunctive normal form is satisfiable,

1. Write as an implication graph.
2. Use DFS or Kosaraju's algorithm to find the strongly connected components.
3. Check whether any strongly-connected component contains both a variable and its negation.

Each of these steps can be done in polynomial time, so the whole process can be done in polynomial time, so 2-CNF-SAT $\in$ P.

4.1.4 Traveling Salesman Problem

TSP as a decision problem

$$\text{TSP} = \{(G, c, k): \begin{array}{l} G = (V, E) \text{ is a complete graph.} \\ c \text{ is a cost function from } V \times V \to \mathbb{N} \\ k \in \mathbb{N}, \text{ and} \\ G \text{ has a traveling-salesman tour with cost at most } k. \end{array} \}$$

The traveling-salesman tour is a Hamiltonian cycle, visiting each node exactly once and returning to the starting point.

To show that TSP $\in$ NP-complete, (a) show that TSP $\in$ P by walking through what it would take to confirm a solution and saying that it can be done in polynomial time, and (b) show that HAM-CYCLE $\leq_P$ TSP by saying that HAM-CYCLE is a special case of TSP. Since HAM-CYCLE is NP-complete, TSP is also.

Let $G = (V, E)$ be an instance of HAM-CYCLE. Form a complete graph $G' = (V, E')$ where $E' = \{(i, j) : i, j \in V \text{ and } i \neq j\}$. Define the cost function by $c(i, j) = 0$ if $(i, j) \in E, 0$ if $(i, j) \notin E$. The instance of TSP is $(G', c, 0)$.

The graph G is a hamiltonian cycle iff graph G' has a tour of cost at most 0.

Triangle Inequality

If the cost function has the property that cutting out an intermediate stop never increases the cost, *i.e.* the most direct path between two points is not more expensive than a meandering one, *i.e.* the triangle inequality holds on the cost function, $c(u, w) \leq c(u, v) + c(v, w)$, the problem is still NP-complete, but we can use Prim's algorithm for a minimal spanning tree to find, in polynomial time, an approximate solution whose total cost is no more than twice the optimal solution.

S19 #S6

Many problem have been proved to be NP-complete. To prove NP-completeness, it is necessary in general to demonstrate two proof components. What are the two proof components to show a problem being NP-complete?

Being NP-complete, the traveling-salesman problem (TSP) has a 2-approximation solution in polynomial time based on establishing a minimum spanning tree (MST) rooted at the start/end vertex (in polynomial time following MST-PRIM), if the graph edge weights observe the triangle inequality. Sketch a brief proof to demonstrate that that such a proof satisfies 2-approximation.

4.1.5 Knapsack Problem

S18 #L4

a. Define the following classes of a decision problem: P, NP, and NP-completeness.
b. Consider the 0-1 knapsack problem with n objects each with its respective pre-defined profit. The objective is to maximize the total profit that can be accommodated into a container of capacity W. Define appropriate notations for weight and profit of objects, formulate the problem.
c. Convert of the problem that you have defined in (b) into a decision problem.
d. Show the problem that you have defined in (c) belongs to NP-class.
e. Does the problem in (d) belong to the P-class or NP-completeness. (Justify your answer.)
f. If principle of optimality be applicable to solve the problem defined in (c), formulate it. Otherwise, explain why not.
g. What would be your explanation, if 0-1 knapsack problem is solved by dynamic programming in polynomial time?

4.1.6 Old Exam Problems

S15 #L1

1. Suppose you have to find a solution to a problem that belongs to NP-complete class. Clearly summarize the steps that will help you to find the solution of the problem.
2. John, an undergraduate student, recently took data structure course, states that heuristics algorithms always solve NP-complete problems. He cites simplex methods as an example. If you agree with John, justify why or why not.
3. What is the strategy behind a greedy algorithm? Will it always provide an optimal solution? If yes explain why, otherwise say why not.
4. Consider the following pseudo code for Kruskal's algorithm for solving minimal spanning tree (MST).
   Algorithm MST. Let N be the number of nodes in graph G.

```
1 Sort the edges in non-decreasing order of cost.
2 T is an empty graph
3 while T has fewer than N-1 edges do:
4         let e denote the next edge of G (in the order of cost)
5         if T∪{e} does not contain a cycle, then T=T∪{e}
```

Clearly mentioning the data structure you have to employ to reduce the time complexity to access and to maintain the necessary information, show the exact time taken to obtain the MST. Also show the tight bound of the algorithm. (Pay attention in detecting a cycle.)

Solutions

1. [From §35, page 1106] There are three ways to approach an NP-Complete problem.

a. If the problem size is sufficiently small that, even if the growth is exponential or factorial, the time and resources required to solve it directly are within the time and money budgets, you may solve it directly.

b. See if the instances you are considering form a special case of the problem that can be solved in polynomial time.

c. Find an approximate solution.

S15 #L3

1. Compare and contrast P, NP, NP-Complete, and NP-Hard.
2. Based on current conjecture, draw a Venn diagram to show the relationship among these classes of problem.
3. Clearly state what is understood by propositional satisfiability problem.
4. Suppose there are m clauses and k propositions in a given 3-p sat problem. How many possible interpretations are there? What is the time complexity of testing the satisfiability of a given interpretation? What is the time and space complexity of testing the satisfiability of the clauses?
5. Suppose you came across a research paper that highlights a clever heuristic strategy that the authors have used to solve an instance of 3-p sat which has 10,000 propositions and 10,000 clauses with 3 propositions. They have generated several instances of the similar compositions of variables and clauses. The authors, in their point of view, have demonstrated the power of their heuristic to solve an instance of NP-complete problem in polynomial time (empirically shown). What would be your insight of their findings?

Solutions

1. **P** Solvable in polynomial time

   NP Verifiable in polynomial time

   NP-Complete Verifiable in polynomial time and at least as hard to solve as any other problem in NP

   NP-Hard At least as hard to solve as any other problem in NP, but not necessarily verifiable in polynomial time.

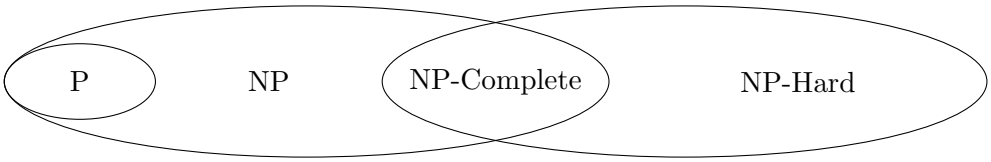
   P $\subset$ NP

   NP-Complete $\subset$ NP

NP-Complete $\subset$ NP-Hard

NP-Complete = NP $\cap$ NP-Hard

The open question is whether there are problems verifiable in polynomial time that are not solvable in polynomial time.

2. 

3. A propositional satisfiability problem asks, given a boolean combinational circuit composed of some number of inputs, AND, OR, and NOT gates, and single output, is there a set of inputs that will yield a TRUE (1) output?

F15 #S5

Mark True or False for each of the following statements.

a. Greedy strategy sometimes finds the best or the optimal solution.
b. Dynamic programming will always find the optimal solution even when the principal of optimality condition is not satisfied.
c. (Same as S5 #S9b) Breadth first search is a special case of heuristic search algorithm.
d. Some problems that belong to NP-class can be solved polynomially.
e. Satisfiability problem of propositional calculus is NP-complete.

Solutions

a.
b.
c.
d. True. Being NP means they can be verified in polynomial time. It does not mean they cannot be solved in polynomial time.
e. True. There are some special cases, like 2-CNF, but in general, SAT is NP-complete.

F16 #S5

Suppose there are n clauses and m variables (propositions) in a given $3-p$ sat problem.

- How many possible interpretations are there?
- Find the tight bound of checking for satisfiability of the n clauses.

F16 #L1

1. Briefly describe NP-class, P-class, NP-complete, and NP-hard.

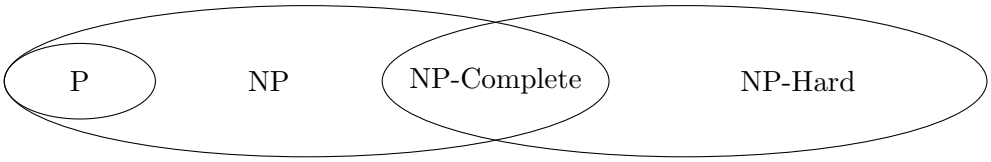
2. Show the conjectured relationship among the classes NP-class, P-class, NP-complete, and NP-hard.
3. Show that counting n objects with integer key values belongs to NP-class.
4. Provide the steps involved in showing whether a problem belongs to NP-complete or not.
5. Illustrate the steps in step d by showing 3 proposition satisfiability (3-p sat) problem belongs to NP-complete.
6. Provide a pseudo code that attempt to solve 3-p sat problem heuristically.

S17 #L2

a. Compare and contrast P, NP, NP-complete, and NP-hard.
b. Based on current conjecture, draw a Venn diagram to show the relationship among these classes of problem.
c. Suppose there n clauses and m propositions in a given 3p-sat problem. How many possible interpretations are there? What is the time complexity of testing the satisfiability of a given interpretation? What is the time and space complexity of testing the satisfiability of the clauses?
d. 3-p sat problem is NP-complete, but people still have published papers by applying heuristics strategy and showing that they were able to solve it with large number of distinct propositions (say 100) and large number of clauses (say 200). To avoid any bias, they have generated the clauses and the set of propositions randomly. How would you start investigating their results? Can their results be generalized?
e. Suppose a single NP-complete problem is solved in polynomial algorithm, what can you state about the entire NP-complete class as well as the NP-hard class.

Solutions

a. **P** Solvable in polynomial time. $O(n^k)$ for constant k.
   NP Verifiable in polynomial time. May or may not be solvable in polynomial time.
   NP-complete Verifiable in polynomial time, but at least as hard to solve as any problem in NP. Conjectured to not have a polynomial-time solution.
   NP-hard Both verification and solution are as hard as for any problem in NP. Not known to be verifiable or solvable in polynomial time. Conjectured to have polynomial-time methods for neither verification nor solution.
   P $\subset$ NP
   NP-complete $\subset$ NP
   NP-complete = NP $\cap$ NP-hard
b.



c. I assume by "number of propositions" they mean "number of propositional variables" or "number of variables." I don't find this topic in the textbook.
The number of interpretations (instances?) is 2^m.
The truth table has 2^m rows and $m + n + 1$ columns. The first m columns are just loop assignment. The next n columns evaluate the 3-disjunction clauses, each of which should take constant time. The last column evaluates the conjunction of the n clauses for each instance, which should take time proportional to the number n of clauses.
So setting up and evaluating each row (interpretation, instance) should take $O(m+n+n) = O(m+n)$. Since $m/2 \leq n$, we can write that as $O(n)$. Since there are 2^m rows, our total time is $O(n \cdot 2^n)$. The memory required for testing the satisfiability of a clause is also $O(n)$.

d.

e. A problem is NP-complete if it is at least as difficult as every problem in NP, *i.e.* a problem A is NP-complete if, for every problem $B \in$ NP, there is a polynomial-time transformation f such that any instance $\beta \in B$ can be transformed into an instance of A, so that $f(\beta) = \alpha \in A$. The problem A is at least as hard as B because, if it were easier, we could use the solution method for A to solve B.
If a single NP-complete problem, like A, is solved in polynomial time, then each problem $B \in$ NP could be solved in polynomial time by transforming (in polynomial time) each instance of B into an instance of A. Thus, if one NP-complete problem could be solved in polynomial time, each and every NP problem (including each NP-complete problems) could be solved in polynomial time.
Finding a polynomial-time solution to an NP-complete problem would have no similar effect on the NP-hard problems that are not NP. The definition of NP-hard says that each of those problems is at least as hard as each of the NP problems, but does not specify that it is at least as hard as any other NP-hard problems. Taking down one does not make the group fall.

S18 #L4

a. Define the following classes of a decision problem: P, NP, and NP-completeness.
b. Consider the 0-1 knapsack problem with n objects each with its respective pre-defined profit. The objective is to maximize the total profit that can be accommodated into a container of capacity W. Define appropriate notations for weight and profit of objects, formulate the problem.
c. Convert of the problem that you have defined in (b) into a decision problem.
d. Show the problem that you have defined in (c) belongs to NP-class.
e. Does the problem in (d) belong to the P-class or NP-completeness. (Justify your answer.)
f. If principle of optimality be applicable to solve the problem defined in (c), formulate it.

Otherwise, explain why not.

g. What would be your explanation, if 0-1 knapsack problem is solved by dynamic programming in polynomial time?

Solutions

a. **P** Solvable in polynomial time
NP Verifiable in polynomial time
NP-complete Verifiable in polynomial time, but at least as hard to solve as any other problem in NP. Conjectured to not have a solution in polynomial time.
P $\subset$ NP, NP-complete $\subset$ NP.

S19 #S6

Many problem have been proved to be NP-complete. To prove NP-completeness, it is necessary in general to demonstrate two proof components. What are the two proof components to show a problem being NP-complete?

Being NP-complete, the traveling-salesman problem (TSP) has a 2-approximation solution in polynomial time based on establishing a minimum spanning tree (MST) rooted at the start/end vertex (in polynomial time following MST-PRIM), if the graph edge weights observe the triangle inequality. Sketch a brief proof to demonstrate that that such a proof satisfies 2-approximation.

Solution

First Part NP-complete problems are the intersection of two sets, NP problems and NP hard problems. To prove that a problem is NP-complete, we show that it belongs to both sets.
To prove that a problem is NP-complete, there are two steps. First, we show that it is NP, meaning that a solution can be verified in polynomial time. Second, we show that it is NP-hard, meaning that it is at least as hard to solve as a known (or conjectured) NP-hard problem.

Second Part

4.2 Traveling Salesman Problem

4.2.1 With the Triangle Inequality

1. Use MST-Prim to find a minimal spanning tree, T.
2. Make a list, H, of the vertices in the order in which they were first visited.
3. This Hamiltonian cycle has cost no more than twice the cost of the optimal cycle.
4. Since MST-Prim is $O(V^2)$, this method is a polynomial-time 2-approximation algorithm.

4.2.2 Old Exam Problems

Spring 2019, #S6

Many problem have been proved to be NP-complete. To prove NP-completeness, it is necessary in general to demonstrate two proof components. What are the two proof components to show a problem being NP-complete?

Being NP-complete, the traveling-salesman problem (TSP) has a 2-approximation solution in polynomial time based on establishing a minimum spanning tree (MST) rooted at the start/end vertex (in polynomial time following MST-PRIM), if the graph edge weights observe the triangle inequality. Sketch a brief proof to demonstrate that that such a proof satisfies 2-approximation.

Solution to Second Part

1. Let $H*$ denote an optimal tour, and $c(H*)$ be the total cost of the edges of $H*$.
2. We obtained the spanning tree T by deleting an edge from any tour, so $c(T) \leq c(H*)$.
3. A full walk, W, of T traverses each edge of T exactly twice, so $c(W) = 2c(T)$, so $c(W) \leq 2c(H*)$.
4. The full walk, W, may not be a tour, because it may visit a vertex more than once. Deleting a vertex from the walk does not increase its cost, because of the triangle inequality. If we delete the second (or later) appearance of a vertex from the walk, we get the preorder walk of the tree T.
5. Let H be the cycle corresponding to the preorder walk. It is Hamiltonian, because the each vertex appears in the preorder walk exactly once. Since $c(H) \leq c(W)$, we get $c(H) \leq 2c(H*)$; therefore, the method satisfies 2-approximation.

Chapter 5

Graph Algorithms

5.1 Graph Algorithms Summary

$O(V+E)$	BFS	Breadth First Search, finds distance from source to each node in an unweighted graph, directed or undirected. Each vertex has a color, a distance, and a parent.
$O(V+E)$	DFS	Depth First Search on an unweighted graph. Each vertex has a color, a discovery time, a finishing time, and a parent. Depth First Tree has tree edges, forward edges, back edges, and cross edges. Used for topological sort.
$O(V^2), O(E \lg V)$	Dijkstra	Finds shortest distance from the source in a weighted graph (directed or undirected). Time complexity depends on the data structure we use for the queue.

5.2 Breadth First Search and Depth First Search (BFS and DFS)

5.2.1 Old Exam Questions

Spring 2015 #S9

Mark true/false against the following statements.

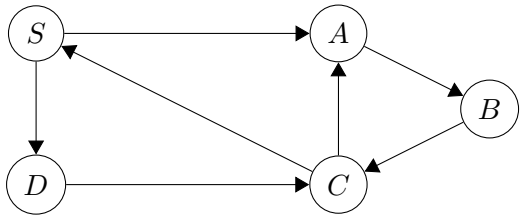
1. (Same as F15 #S7a) A binary search tree of size N will always find a key in at most $O(\log N)$ time.
2. (Same as F15 #S5c) A breadth first search algorithm can be considered as a special case of heuristic search algorithm.
3. An optimal binary search tree is not necessarily a balanced tree.
4. A dynamic programming approach uses top-down problem solving strategy to solve optimization problem.

Solutions

1. Yes. $\log N = h$, the height of the tree.
2. No. (? The textbook doesn't talk about heuristics.)
3. An *optimal binary search tree* is a tree whose expected cost is least. We compute the expected cost using the (probability of searching for each node) $\times$ (depth of that node).
   A *balanced tree* is one such that no two leaves differ in height by more than one.
   I think the answer is "No," because an optimal binary search tree has to be weight-balanced, not height-balanced.
4. No. Dynamic programming uses bottom-up.

Spring 2019 #S5

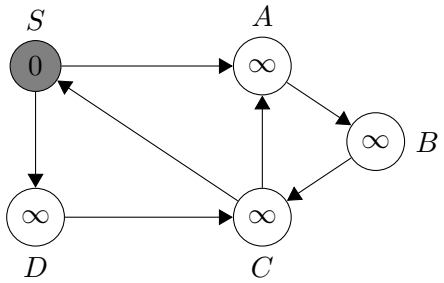
BFS (breadth-first search) and DFS (depth-first search): Give the visited node order for each type of graph search, starting with S below.



Solution

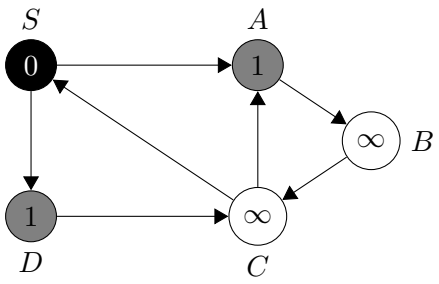
BFS: $SADBC$ or $SDACB$, depending on the order in which the vertices are listed in V.

Method for $V = \{S, A, B, C, D\}$, giving visited node order $SADBC$.



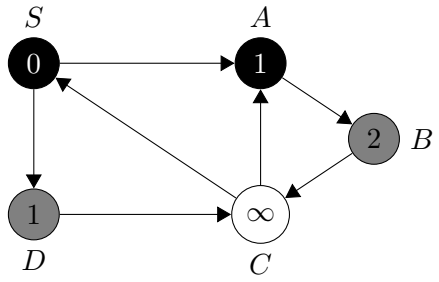
$Q = \{S\}$

u	S	A	B	C	D
$u.color$	$Gray$	$White$	$White$	$White$	$White$
$u.d$	0	∞	∞	∞	∞
$u.\pi$	NIL	NIL	NIL	NIL	NIL



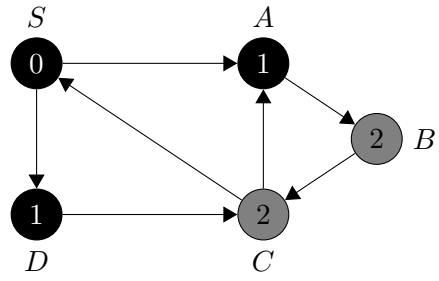
$Q = \{A, D\}$

u	S	A	B	C	D
$u.color$	$Black$	$Gray$	$White$	$White$	$Gray$
$u.d$	0	1	∞	∞	1
$u.\pi$	NIL	S	NIL	NIL	S



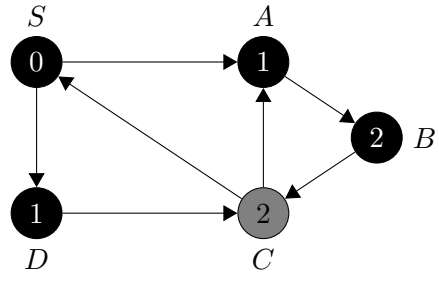
$Q = \{D, B\}$

u	S	A	B	C	D
$u.color$	$Black$	$Black$	$Gray$	$White$	$Gray$
$u.d$	0	1	2	∞	1
$u.\pi$	NIL	S	A	NIL	S



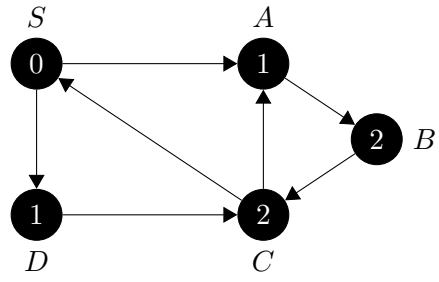
$Q = \{B, C\}$

u	S	A	B	C	D
$u.color$	$Black$	$Black$	$Gray$	$Gray$	$Black$
$u.d$	0	1	2	2	1
$u.\pi$	NIL	S	A	D	S



$Q = \{C\}$

u	S	A	B	C	D
$u.color$	$Black$	$Black$	$Gray$	$Gray$	$Black$
$u.d$	0	1	2	2	1
$u.\pi$	NIL	S	A	D	S

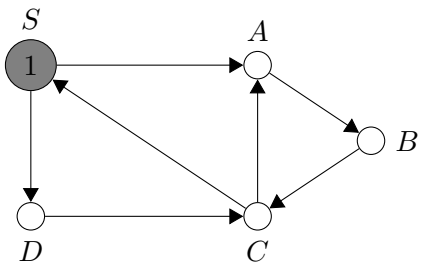


$Q = \{\}$

u	S	A	B	C	D
$u.color$	$Black$	$Black$	$Gray$	$Gray$	$Black$
$u.d$	0	1	2	2	1
$u.\pi$	NIL	S	A	D	S

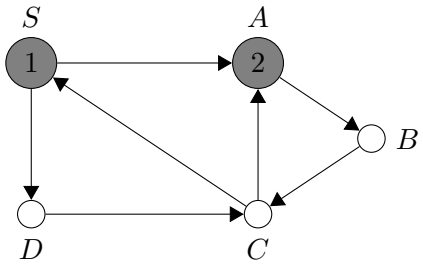
DFS: $SABCD$ or $SDCAB$, depending on the order of V.

$V = \{S, A, B, C, D\}$



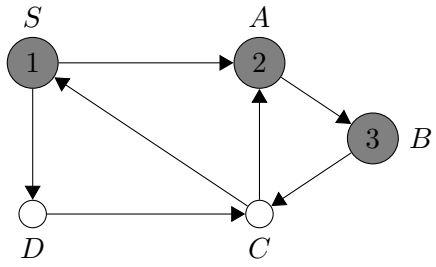
DFS-Visit (G, S), $time = 1$

u	S	A	B	C	D
$u.color$	$Gray$	$White$	$White$	$White$	$White$
$u.d$	1				
$u.f$					
$u.\pi$	NIL	S	NIL	NIL	NIL



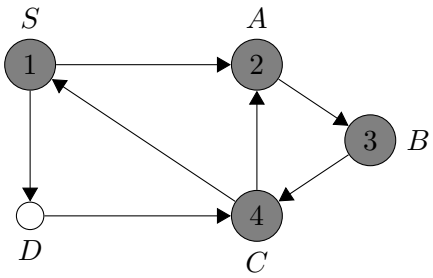
DFS-Visit (G, A), $time = 2$

u	S	A	B	C	D
$u.color$	$Gray$	$Gray$	$White$	$White$	$White$
$u.d$	1	2			
$u.f$					
$u.\pi$	NIL	S	A	NIL	NIL



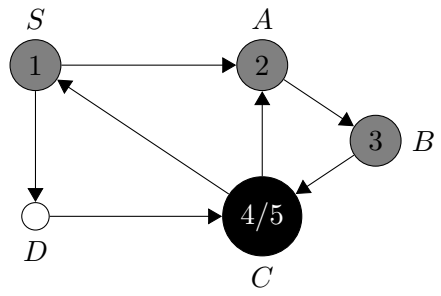
DFS-Visit (G, B), $time = 3$

u	S	A	B	C	D
$u.color$	$Gray$	$Gray$	$Gray$	$White$	$White$
$u.d$	1	2	3		
$u.f$					
$u.\pi$	NIL	S	A	B	NIL



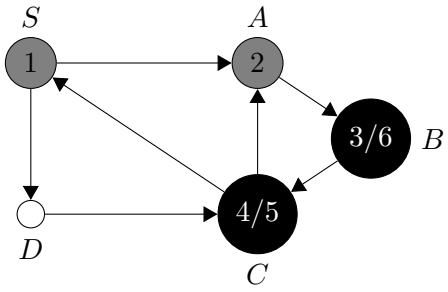
First part of DFS-Visit (G, C), $time = 4$

u	S	A	B	C	D
$u.color$	$Gray$	$Gray$	$Gray$	$Gray$	$White$
$u.d$	1	2	3	4	
$u.f$					
$u.\pi$	NIL	S	A	B	NIL



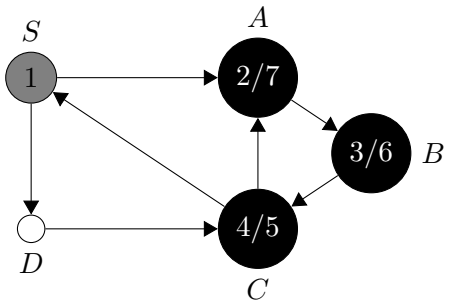
Second part of DFS-Visit (G, C), $time = 5$

u	S	A	B	C	D
$u.color$	$Gray$	$Gray$	$Gray$	$Black$	$White$
$u.d$	1	2	3	4	
$u.f$				5	
$u.\pi$	NIL	S	A	B	NIL



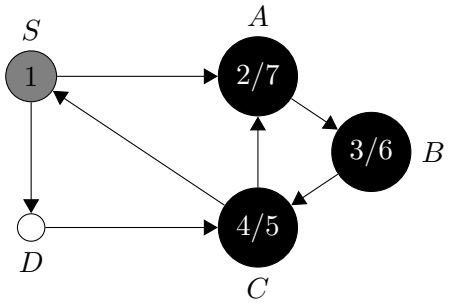
Back to second part of DFS-Visit (G, B), $time = 6$

u	S	A	B	C	D
$u.color$	$Gray$	$Gray$	$Black$	$Black$	$White$
$u.d$	1	2	3	4	
$u.f$			6	5	
$u.\pi$	NIL	S	A	B	NIL



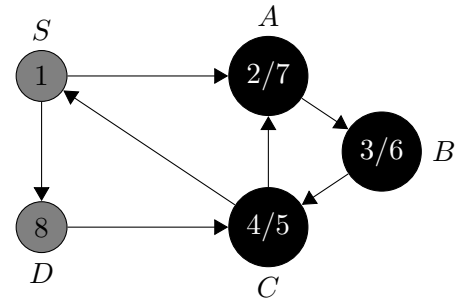
Back to second part of DFS-Visit (G, A), $time = 7$

u	S	A	B	C	D
$u.color$	$Gray$	$Black$	$Black$	$Black$	$White$
$u.d$	1	2	3	4	
$u.f$		7	6	5	
$u.\pi$	NIL	S	A	B	NIL



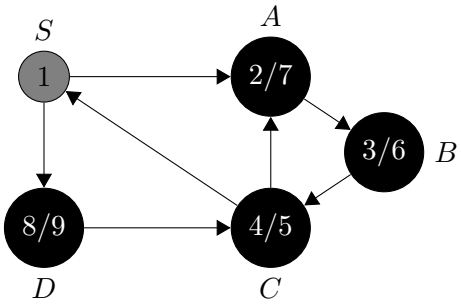
Back to `for` loop in DFS-Visit (G, S), $time = 7$

u	S	A	B	C	D
$u.color$	$Gray$	$Black$	$Black$	$Black$	$White$
$u.d$	1	2	3	4	
$u.f$		7	6	5	
$u.\pi$	NIL	S	A	B	S



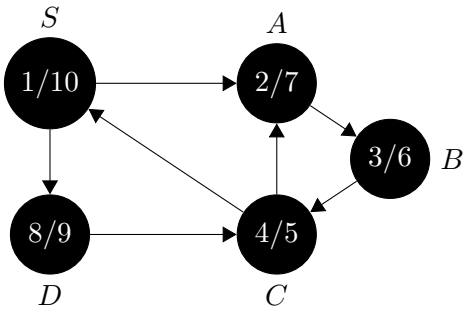
First part of DFS-Visit (G, D), $time = 8$

u	S	A	B	C	D
$u.color$	$Gray$	$Black$	$Black$	$Black$	$Gray$
$u.d$	1	2	3	4	8
$u.f$		7	6	5	
$u.\pi$	NIL	S	A	B	S



Second part of DFS-Visit (G, D), $time = 9$

u	S	A	B	C	D
$u.color$	$Gray$	$Black$	$Black$	$Black$	$Black$
$u.d$	1	2	3	4	8
$u.f$		7	6	5	9
$u.\pi$	NIL	S	A	B	S



Second part of DFS-Visit (G, S), $time = 10$

u	S	A	B	C	D
$u.color$	$Gray$	$Black$	$Black$	$Black$	$Black$
$u.d$	1	2	3	4	8
$u.f$	10	7	6	5	9
$u.\pi$	NIL	S	A	B	S

Fall 2015 #S5

Mark True or False for each of the following statements.

a. Greedy strategy sometimes finds the best or the optimal solution.
b. Dynamic programming will always find the optimal solution even when the principal of optimality condition is not satisfied.
c. (Same as S5 #S9b) Breadth first search is a special case of heuristic search algorithm.
d. Some problems that belong to NP-class can be solved polynomially.
e. Satisfiability problem of propositional calculus is NP-complete.

Solutions

a. True.
b. False.
c. Don't know.

d. True.
e. True.

Fall 2016 #S8

Mark true/false (T/F) against the following statements.

1. A binary search tree of size N will always find a key at most $O(\lg N)$ time
2. A breadth first search can be considered as a special case of heuristic search algorithm.
3. An optimal binary search tree is not necessarily being a balanced tree.
4. A dynamic programming approach uses top-down problem solving strategy to solve optimization problem.

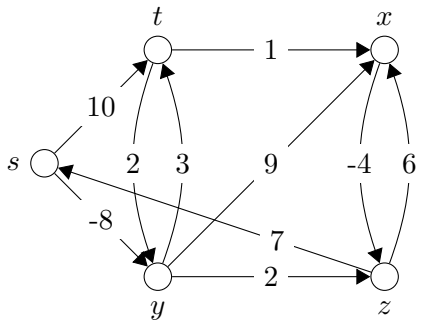
Solutions

1. True
2. Don't know.
3. True.
4. False.

Fall 2018 #L3

Follow depth-first search (DFS), starting from Node s, to traverse the graph shown below, with its edge weights all ignored and the start time equal to 1. Mark (1) the type of every edge and (2) the discovery and finish times of each node.

Follow breadth-first search (BFS), starting from Node s, to traverse the graph shown below, with its edge weights all ignored. Show the predecessor tree rooted at Node s after BFS, with the number of links (i.e. distance) from Node s to every other node indicated.



[Note: If the edge weights aren't relevant, why did they leave them in the diagram?]

DFS Solution

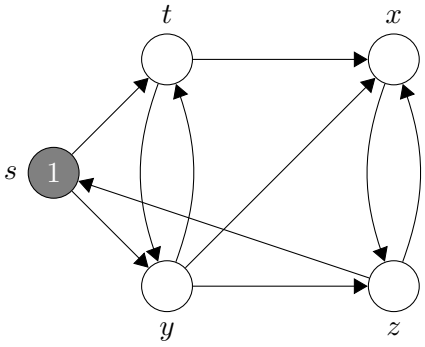
There are four types of edges.

Tree edges are edges in the depth-first forest G_π. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v). The tree edges below are in bold.

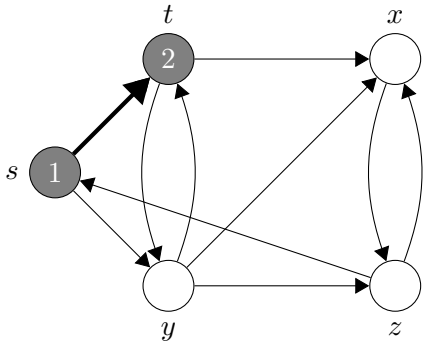
Back edges connect a vertex to an ancestor.

Forward edges are non-tree edges that connect a vertex to a descendent.

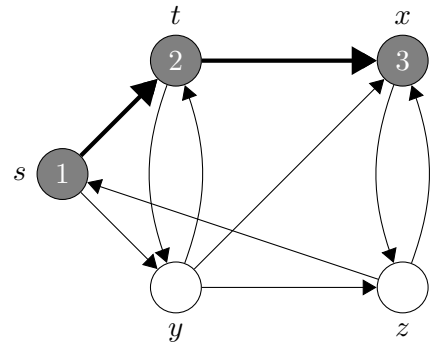
Cross edges are everything else.



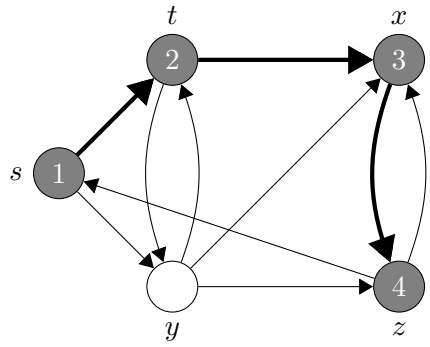
c	s	t	x	y	z
$c.color$	$Gray$	$White$	$White$	$White$	$White$
$c.d$	1				
$c.f$					
$c.\pi$	NIL	NIL	NIL	NIL	NIL



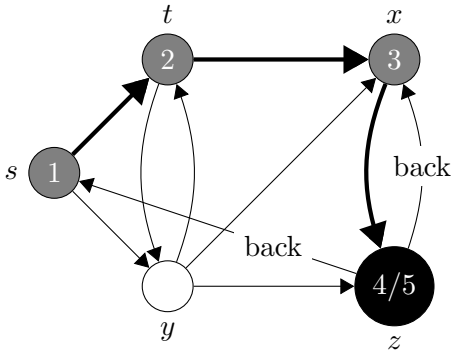
c	s	t	x	y	z
$c.color$	$Gray$	$Gray$	$White$	$White$	$White$
$c.d$	1	2			
$c.f$					
$c.\pi$	NIL	S	NIL	NIL	NIL



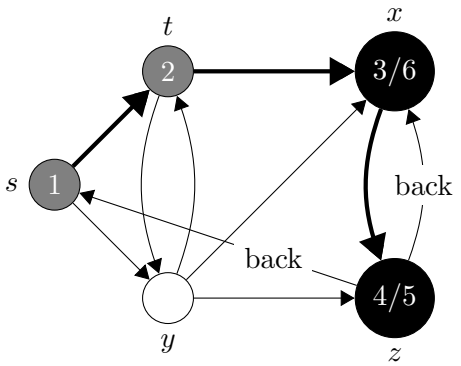
c	s	t	x	y	z
$c.color$	$Gray$	$Gray$	$Gray$	$White$	$White$
$c.d$	1	2	3		
$c.f$					
$c.\pi$	NIL	s	t	NIL	NIL



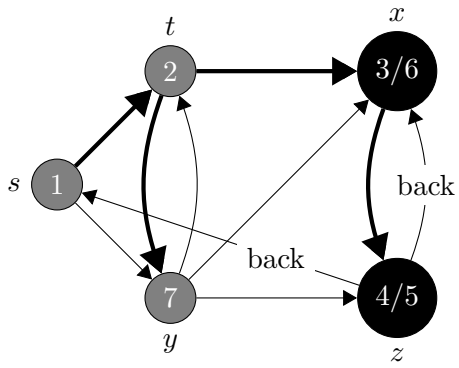
c	s	t	x	y	z
$c.color$	$Gray$	$Gray$	$Gray$	$White$	$Gray$
$c.d$	1	2	3		4
$c.f$					
$c.\pi$	NIL	s	t	NIL	x



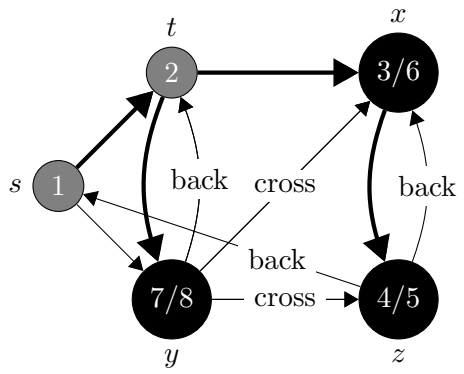
c	s	t	x	y	z
$c.color$	$Gray$	$Gray$	$Gray$	$White$	$Black$
$c.d$	1	2	3		4
$c.f$					5
$c.\pi$	NIL	s	t	NIL	x



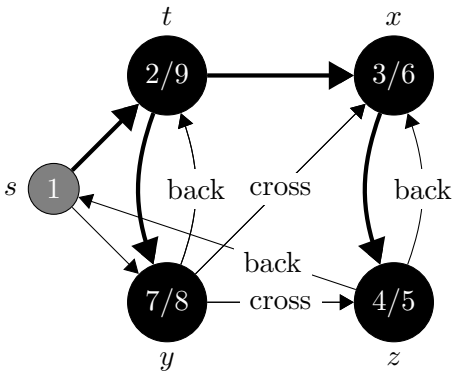
c	s	t	x	y	z
$c.color$	$Gray$	$Gray$	$Black$	$White$	$Black$
$c.d$	1	2	3		4
$c.f$			6		5
$c.\pi$	NIL	s	t	NIL	x



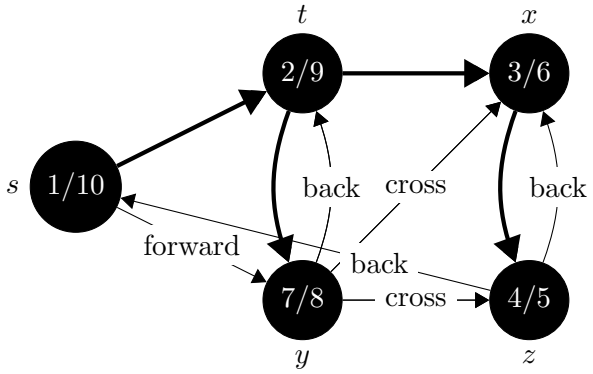
c	s	t	x	y	z
$c.color$	$Gray$	$Gray$	$Black$	$Gray$	$Black$
$c.d$	1	2	3	7	4
$c.f$			6		5
$c.\pi$	NIL	s	t	t	x



c	s	t	x	y	z
$c.color$	$Gray$	$Gray$	$Black$	$Black$	$Black$
$c.d$	1	2	3	7	4
$c.f$			6	8	5
$c.\pi$	NIL	s	t	t	x

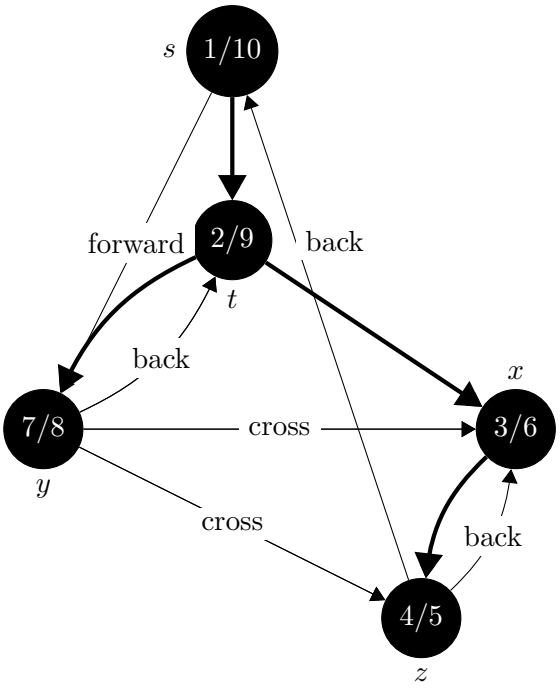


c	s	t	x	y	z
$c.color$	$Gray$	$Black$	$Black$	$Black$	$Black$
$c.d$	1	2	3	7	4
$c.f$		9	6	8	5
$c.\pi$	NIL	s	t	t	x



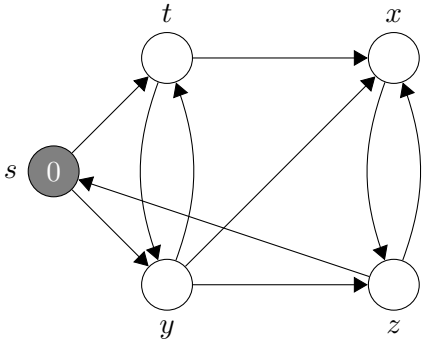
c	s	t	x	y	z
$c.color$	$Black$	$Black$	$Black$	$Black$	$Black$
$c.d$	1	2	3	7	4
$c.f$	10	9	6	8	5
$c.\pi$	NIL	s	t	t	x

Depth First Tree



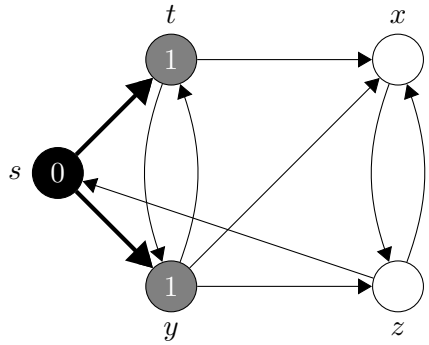
BFS Solution

$V = \{s, t, x, y, z\}$



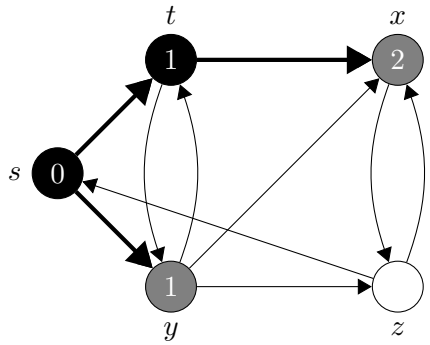
$Q = \{s\}$

c	s	t	x	y	z
$c.color$	$Gray$	$White$	$White$	$White$	$White$
$c.d$	0				
$c.\pi$	NIL	NIL	NIL	NIL	NIL



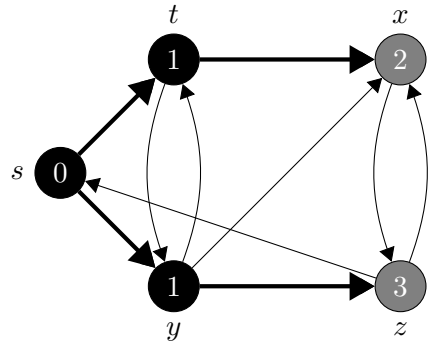
$Q = \{t, y\}$

c	s	t	x	y	z
$c.color$	$Black$	$Gray$	$White$	$Gray$	$White$
$c.d$	0	1		1	
$c.\pi$	NIL	s	NIL	s	NIL



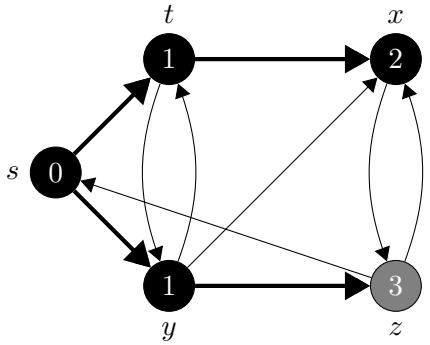
$Q = \{y, x\}$

c	s	t	x	y	z
$c.color$	$Black$	$Black$	$Gray$	$Gray$	$White$
$c.d$	0	1	2	1	
$c.\pi$	NIL	s	t	s	NIL



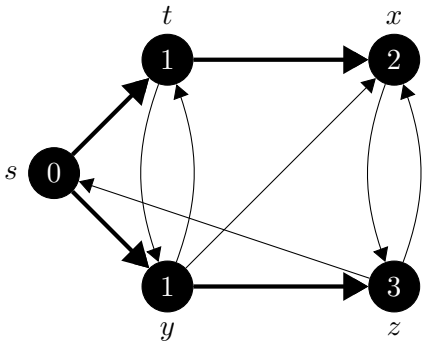
$Q = \{x, z\}$

c	s	t	x	y	z
$c.color$	$Black$	$Black$	$Gray$	$Black$	$Gray$
$c.d$	0	1	2	1	3
$c.\pi$	NIL	s	t	s	y



$Q = \{z\}$

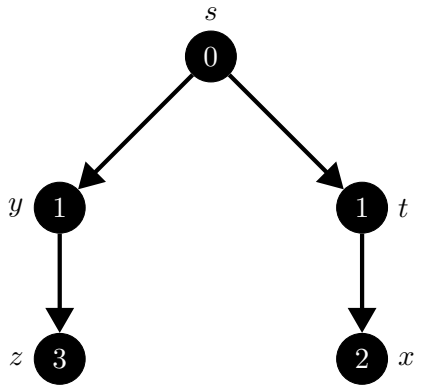
c	s	t	x	y	z
$c.color$	$Black$	$Black$	$Black$	$Black$	$Gray$
$c.d$	0	1	2	1	3
$c.\pi$	NIL	s	t	s	y



$Q = \{\}$

c	s	t	x	y	z
$c.color$	$Black$	$Black$	$Black$	$Black$	$Black$
$c.d$	0	1	2	1	3
$c.\pi$	NIL	s	t	s	y

Predecessor Tree



5.3 Strongly Connected Components

5.3.1 Definition

A **strongly connected component** of a directed graph $G = (V, E)$ is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v in C, we have both $u \rightsquigarrow v$ and $v \rightsquigarrow u$; that is, vertices u and v are reachable from each other, that there are both a path from u to v and a path from v to u.

5.3.2 Algorithm

Strongly-Connected-Components (G)

1. Run DFS(G) to compute finishing times $u.f$ for $u \in V$.
2. Sort V in decreasing order by $u.f$.
3. Run DFS(V^T)
4. Output the vertices of each depth-search tree as a separate strongly connected component.

5.3.3 Old Exam Questions

Spring 2017 #S3

a. Define strongly connected components as applied to directed graphs.
b. Some potential application of strongly connected components.
c. Provide a pseudo code for obtaining strongly components of a directed graph. (Hint: Use depth first search on appropriately transformed graphs.)

Solutions

a. A strongly connected component of a graph $G(V, E)$ is a maximal subset $C \subseteq V$ such that, for each pair (u, v) of vertices in C, there is a path from u to v and a path from v to u. To

find the strongly connected components (plural) of a graph is to partition it into strongly connected components.

b. An application is in determining boolean 2-satisfiability. Another is finding groups in a social network.

c. Strongly-Connected-Components

   1. Run DFS on G.
   2. Sort the vertices of G in decreasing order by their finishing time.
   3. Create G^T, in which all of the edges are reversed.
   4. Run DFS on G^T using the vertices sorted as above.
   5. The trees in the depth-first forest are separate strongly connected components.

Fall 2016 #S7

1. Define strongly connected components.
2. Does a strongly connected component graph cyclic or acyclic? Justify your answer.

Solutions

1. The strongly connected components of a graph $G = (V, E)$ is a partition of the graph into maximal subsets $C_i \subseteq V$ such that, for each C_i, for each pair of vertices (u, v) in C_i, there is a path from u to v and a path from v to u.
2. A strongly connected component graph is cyclic. A directed graph is cyclic iff it contains a cycle. A cycle is a path from a vertex to itself. A strongly connected component graph has, by definition, for each vertex u, a path to another (actually, each other) vertex v and a path back. Connecting these two paths, you get a cycle. Thus, a strongly connected graph has a cycle, and is therefore cyclic.

Fall 2015 #S7

Mark True or False against the following statements.

a. (Same as S15 #9a) A binary search tree of size N will always find a key in at most $O(\log N)$ time.

b. An optimal binary search is not necessary a balanced tree.

c. A binary heap always maintains a balanced tree as practical as it can be.

d. To implement a priority queue binomial heap is preferred over binary heap.

e. A graph formed by strongly connected components, a strongly connected components graph (SCC) is always a minimum spanning tree.

Solutions

a. False. It will find a key in at most $O(h)$ time, where h is the height of the tree. If the tree is balanced, then $h = \lg N$, but the tree does not need to be balanced. In the worst case, the

tree is a single branch, and $h = N$.

b. True.

c. True. The definition of a binary heap says that it is a balanced tree.

d. True. (?)

e. False. It is a spanning tree, but not necessarily minimal.

Spring 2015 #S4

Mark true/false (T/F) against the following statements:

1. Connecting any pair of nodes in a minimum spanning tree will always form a cycle.
2. Building strongly connected component graph of a directed graph of N nodes takes $O(N^2)$ time.
3. A graph formed by strongly connected component nodes, a strongly connected component graph (SCC) is always a minimum spanning tree.
4. SCC graph will be useful in determining articulation node in a graph.

Solutions

1. True.
2. False. Building a strongly connected component graph uses DFS twice. DFS takes $O(V+E)$. If the graph is complete, then $E = V^2$ and the operation takes $O(V^2)$ time, but if the graph is sparse, it runs in faster time.
3. False. It is a spanning tree, but not necessarily minimal.
4. True.

5.4 Minimum Spanning Trees

5.4.1 Definitions

Given a connected, weighted, undirected graph $G = (V, E) \ldots$

Spanning Tree An acyclic subset $T \subseteq E$ that connects all vertices. Since it is acyclic and connects all vertices, it must be a tree.

Minimum Spanning Tree A spanning tree T such that the total weight of the edges is minimized.

$$w(T) = \sum_{(u,v)\in T} w(u, v)$$

We have greedy algorithms that run in $O(E \lg V)$ time if we use binary heaps to store the list of edges.

5.4.2 Kruskal's Algorithm

Create empty set of edges A.

Loop over the edges (u, v) in nondecreasing order:

If u and v are not connected in A:

Add (u, v) to the set A.

Saying that u and v are not connected is the same as saying that adding (u, v) would not create a cycle.

5.4.3 Prim's Algorithm

Pick any starting vertex. At each step, choose the cheapest edge from the set of visited vertices to the set of unvisited vertices. Add that edge to the tree.

5.4.4 Old Exam Questions

Spring 2019 #S6

Many problem have been proved to be NP-complete. To prove NP-completeness, it is necessary in general to demonstrate two proof components. What are the two proof components to show a problem being NP-complete?

Being NP-complete, the traveling-salesman problem (TSP) has a 2-approximation solution in polynomial time based on establishing a minimum spanning tree (MST) rooted at the start/end vertex (in polynomial time following MST-PRIM), if the graph edge weights observe the triangle inequality. Sketch a brief proof to demonstrate that that such a proof satisfies 2-approximation.

Spring 2017 #S6

Briefly describe the minimum spanning tree. Sketch an algorithm to obtain a minimum spanning tree.

Solution

Given an undirected, weighted, connected graph $G(V, E)$, a spanning tree T is a connected acyclic graph that includes all of the vertices V. A minimum spanning tree is a spanning tree with minimum total weight,

$$w(T) = \sum_{(u,v)\in T} w(u, v)$$

We have greedy algorithms for growing a minimum spanning tree.

Kruskal Sort the edges in nonincreasing order. Consider each edge in order. If adding that edge to the tree does not create a cycle, add it.

Prim Choose any starting vertex. At each step, choose the cheapest edge from the set of visited vertices to the set of unvisited vertices.

Fall 2015 #S4

Define minimum spanning tree. What are the salient features of a minimum spanning tree? Name a couple of [two] algorithms that will help to find the minimum spanning tree from a graph. Out of the two, which one would you prefer and specify why.

Solution

A minimum spanning tree of a connected, undirected, weighted graph $G(V, E)$ is a subset of the edges that connect all of the vertices with minimum total weight.

We have two greedy algorithms for growing a minimal spanning tree, Kruskal's and Prim's.

I prefer Prim's because I find it easier to implement and it has other applications, as in finding a 2-approximation for the Traveling Salesman Problem in the case where the weights satisfy the Triangle Inequality, which is to say that the weights form a metric on the vertices.

Spring 2015 #S4

Mark true/false (T/F) against the following statements:

1. Connecting any pair of nodes in a minimum spanning tree will always form a cycle.
2. Building strongly connected component graph of a directed graph of N nodes takes $O(N^2)$ time.
3. A graph formed by strongly connected component nodes, a strongly connected component graph (SCC) is always a minimum spanning tree.
4. SCC graph will be useful in determining articulation node in a graph.

Solutions

1. True.
2. True. Building a strongly connected component graph requires DFS, which takes $O(V + E)$ time. If the directed graph is complete, then $E = V^2$, but if the graph is a tree then $E = V - 1$, so the best-possible case is $\Omega(V)$, but the worst case is $O(V^2)$.
3. False for several reasons.
    a. An SCC is directed and unweighted while an MST is undirected and weighted.
    b. An SCC, if it were undirected, could contain cycles, meaning that it is not a tree.
4. True.

Spring 2015 #L1

1. Suppose you have to find a solution to a problem that belongs to NP-complete class. Clearly summarize the steps that will help you to find the solution of the problem.
2. John, an undergraduate student, recently took data structure course, states that heuristics algorithms always solve NP-complete problems. He cites simplex methods as an example. If you agree with John, justify why or why not.
3. What is the strategy behind a greedy algorithm? Will it always provide an optimal solution? If yes explain why, otherwise say why not.
4. Consider the following pseudo code for Kruskal's algorithm for solving minimal spanning tree (MST).

   Algorithm MST. Let N be the number of nodes in graph G.

```
1 Sort the edges in non-decreasing order of cost.
2 T is an empty graph
3 while T has fewer than N-1 edges do:
4         let e denote the next edge of G (in the order of cost)
5         if T∪{e} does not contain a cycle, then T=T∪{e}
```

   Clearly mentioning the data structure you have to employ to reduce the time complexity to access and to maintain the necessary information, show the exact time taken to obtain the MST. Also show the tight bound of the algorithm. (Pay attention in detecting a cycle.)

Solutions

1. It is not at all certain, nor even likely, that you can find the solution. If the problem size is small, you may be able to run it directly in a short enough time. If the problem is a special case of an NP-complete problem that can be solved in polynomial time, then you didn't really have an NP-complete problem. If there is an algorithm for an approximate solution, that may work. If there is no approximate algorithm, then you have to convince the person who assigned you this problem that nobody can solve the problem; see the cartoons at the beginning of Garey and Johnson (1979).
2. I disagree. Proof by contradiction. If heuristic algorithms, or any algorithms, can always solve NP-complete problems [in finite time], then P = NP, which is generally conjectured to not be true.
3. The strategy behind greedy algorithms is to find optimal solutions to subproblems, then find optimal solutions to combinations of those subproblems until you have solved the original problem. It works if the problem has optimal substructure and the greedy choice property. A problem has the greedy choice property iff a greedy algorithm gives the optimal solution. If the problem does not have the greedy choice property, a greedy algorithm is not guaranteed to give an optimal solution.

   For example, set covering. I have a set X and a set of n subsets S_i such that the union covers

X, $\bigcup_{1 \le i \le n} S_i = X$. To choose a, perhaps, smaller set that covers X, I could use this greedy algorithm. Let $Y = X$ and $T = \emptyset$. Loop through the list of subsets. If a subset S_i contains an element in Y, add the set S_i to T and delete from Y all of the elements in $Y \cap S_i$. Continue until Y is empty. This algorithm gives a covering set, but not a minimal covering set.

4. (The answer to this part is on page 633. It's really complicated.)

5.5 Single-Source Shortest Paths

$O(VE)$	Bellman-Ford	Edge weights may be negative.
$O(V^2)$	Dijkstra	Edge weights are nonnegative

5.6 Dijkstra

BFS gives the single-source shortest path for an unweighted (possibly directed) graph.

Dijkstra's algorithm gives the single-source shortest path for a weighted, directed graph with no edges of negative weight.

The process of relaxing an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u, and, if so, updating $v.d$ and $v.\pi$.

In general, the algorithm takes $O(V^2)$ if the candidate vertices are kept in an array.

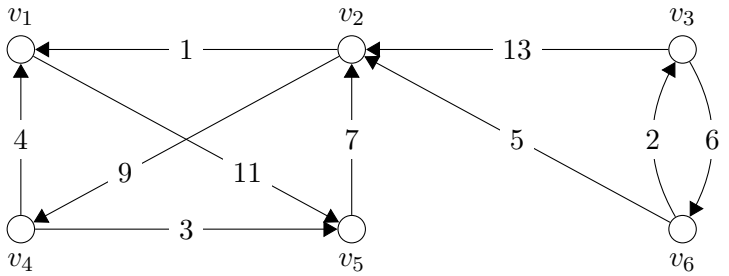
If the graph is sparse, implementing the min-priority queue with a binary min-heap and get it down to $O(E \lg V)$.

5.6.1 Old Exam Questions

Fall 2018 #L4

The Dijkstra's algorithm (DS) solves the single-source shortest-path problem in a weighted graph $G = (V, E)$ without negative weighted edges or cycles, by edge relaxation at one vertex at a time until all vertices are examined. Given the graph G below, follow DS to find the shortest paths from vertex v_1 to all other vertices, with all predecessor edges shaded and estimated distance values from v_1 to all vertices provided at the end. Also list the sequence of vertices at which relaxation takes place.

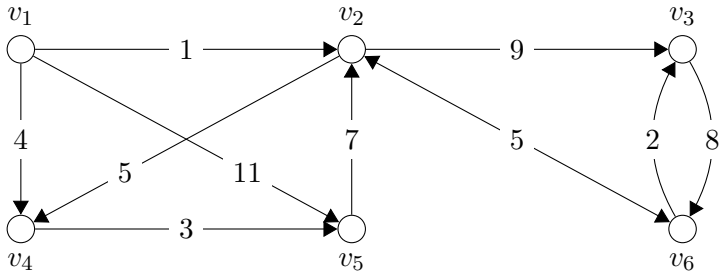
What is the time complexity of DS for a general graph $G = (V, E)$ when candidate vertices are kept in an array?



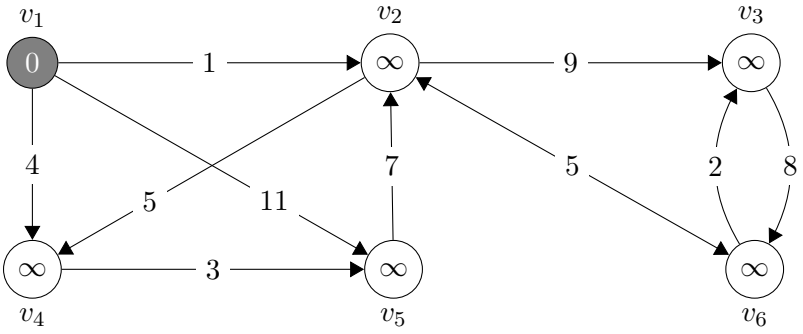
Spring 2018 #S4 [Same as above]

The Dijkstra's algorithm (DS) solves the single-source shortest-path problem in a weighted graph $G = (V, E)$ without negative weighted edges or cycles, by edge relaxation at one vertex at a time until all vertices are examined. Given the graph G below, follow DS to find the shortest paths from vertex v_1 to all other vertices, with all predecessor edges shaded and estimated distance values from v_1 to all vertices provided at the end. Also list the sequence of vertices at which relaxation takes place.

What is the time complexity of DS for a general graph $G = (V, E)$ when candidate vertices are kept in an array?

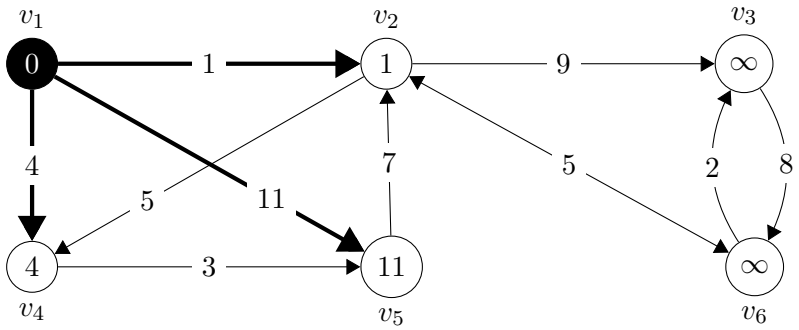


Solution



Initialize-single-source (G, v_1)

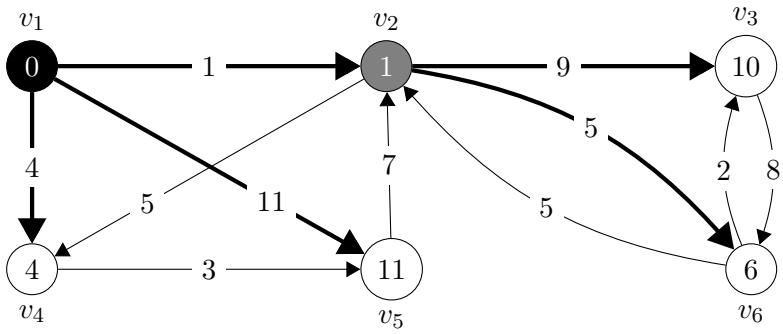
$Q = \{v_1, v_2, v_3, v_4, v_5, v_6\}$



Extract v_1.

$Q = \{v_2, v_3, v_4, v_5, v_6\}$

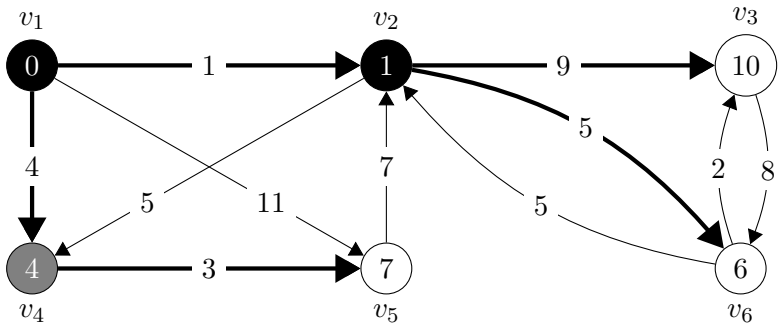
Update v_2, v_4, and v_5.



Extract v_2.

$Q = \{v_3, v_4, v_5, v_6\}$

Update v_3, and v_6.

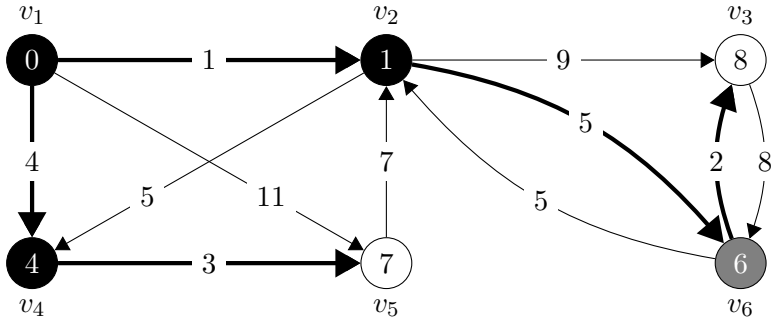


Extract v_4.

$Q = \{v_3, v_5, v_6\}$

Update v_5.

Update the path from v_1 to v_5 to v_1, v_4, v_5

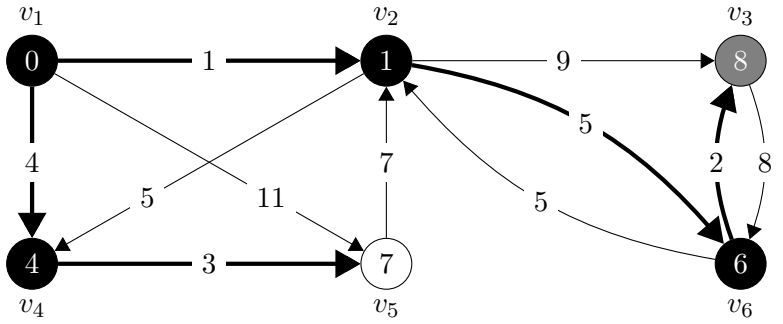


Extract v_6.

$Q = \{v_3, v_5\}$

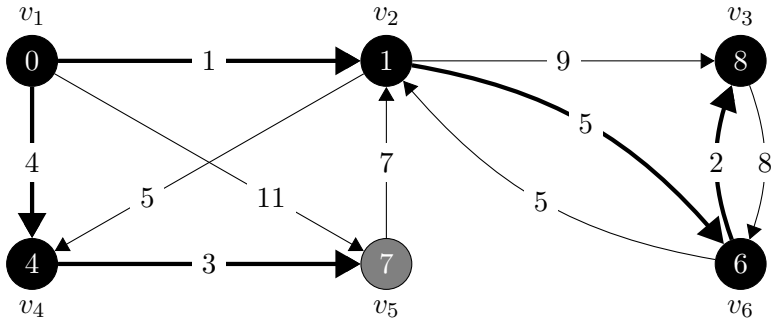
Update v_3.

Update the path from v_1 to v_3 to v_1, v_2, v_6,



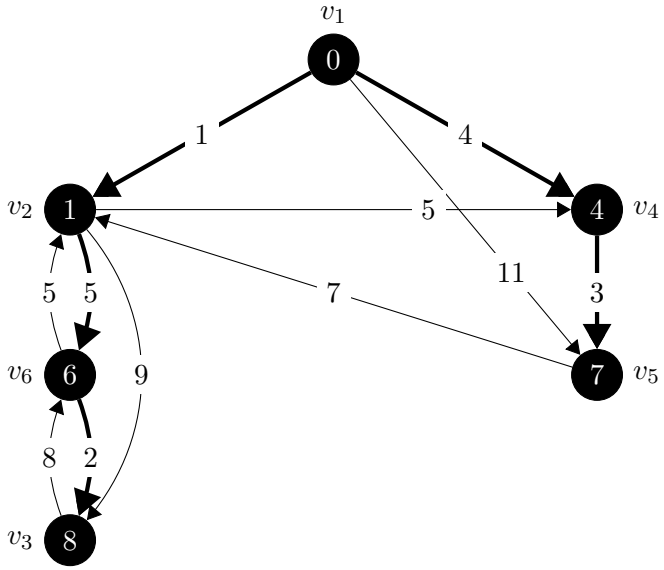
Extract v_3.

$Q = \{v_5\}$



Extract v_5.

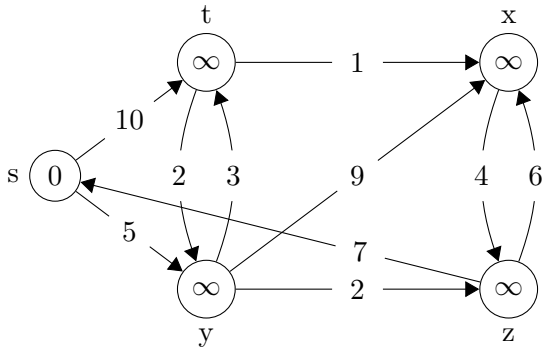
$Q = \{\}$



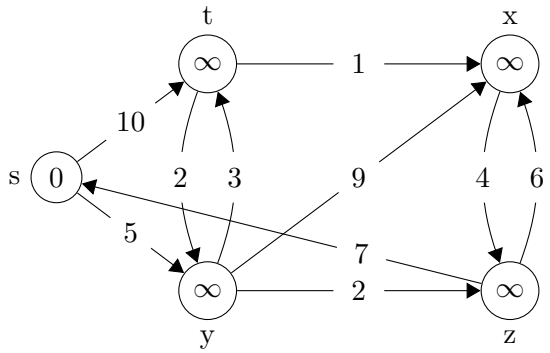
Spring 2019 #L4

The Dijkstra's algorithm (DIJ) solves the single-source shortest-path problem in a weighted directed graph $G = (V, E)$. Given the graph G below, follow DIJ to find shortest paths from vertex s to all other vertices, with all predecessor edges shaded and estimated distance values from s to all vertices provided at the end of each iteration.

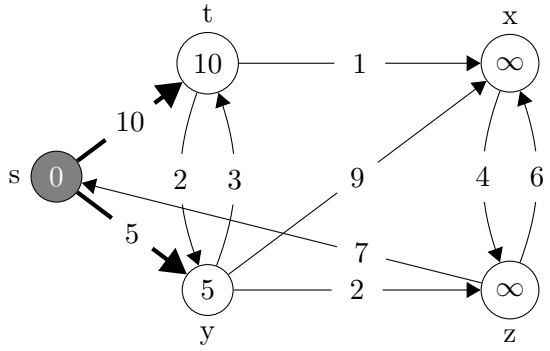
What is the time complexity of DIJ for a general graph $G = (V, E)$, if the candidate vertices are kept in a binary min-heap?



[Note that the diagram is the example in the textbook on page 659.]



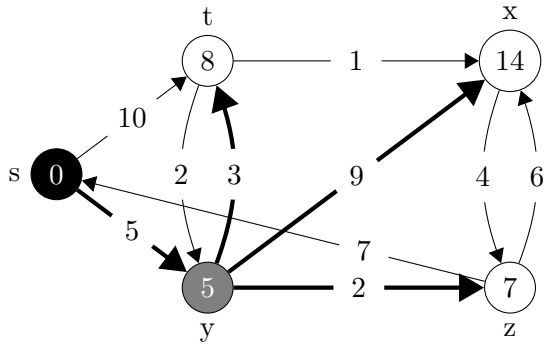
$Q = \{s, t, x, y, z\},\ S = \{\}$



Extract s.

$Q = \{t, x, y, z\},\ S = \{s\}$

Update t and y.

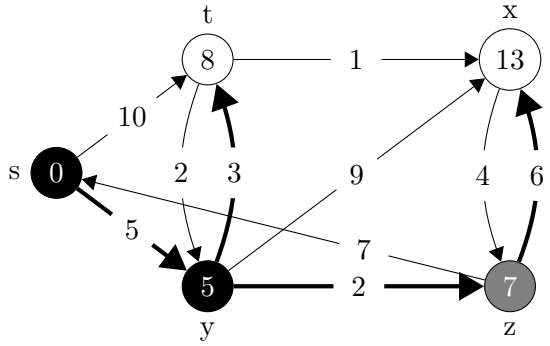


Extract y.

$Q = \{t, x, z\},\ S = \{s, y\}$

Update t, x, and z.

Change the path from s to t from s, t to s, y, t.

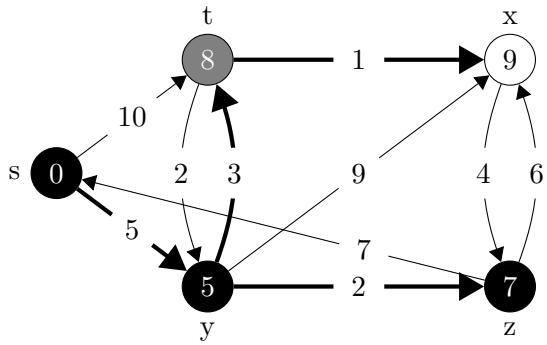


Extract z.

$Q = \{t, x\},\ S = \{s, y, z\}$

Update x.

Change the path from s to x from s, y, x to s, y, z, x.

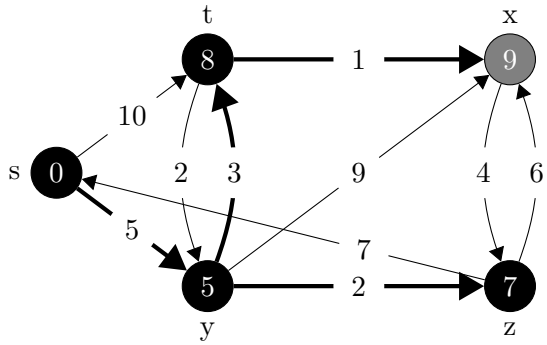


Extract t.

$Q = \{x\},\ S = \{s, t, y, z\}$

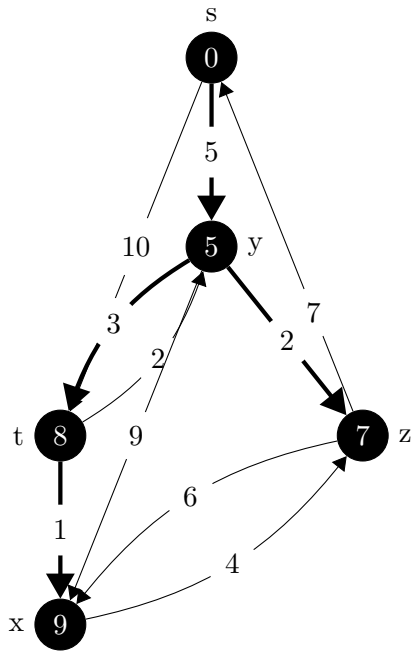
Update x.

Change the path from s to x from s, y, z, x to s, y, t, x.



Extract x.

$Q = \{\},\ S = \{s, t, x, y, z\}$

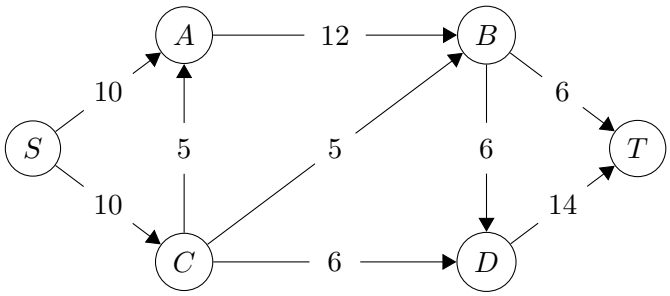


5.7 Ford-Fulkerson Algorithm

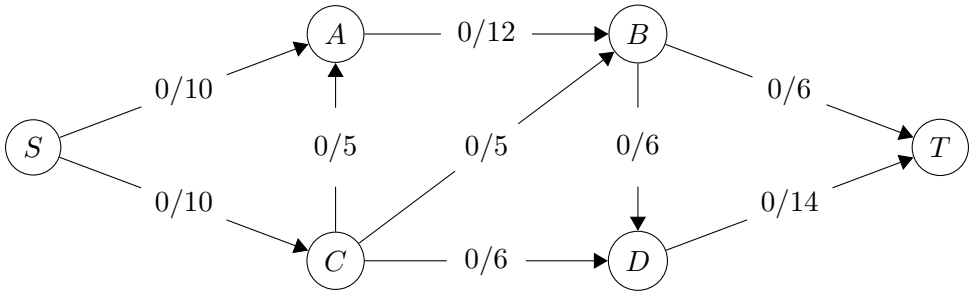
5.7.1 Old Exam Problems

Spring 2019 #L3

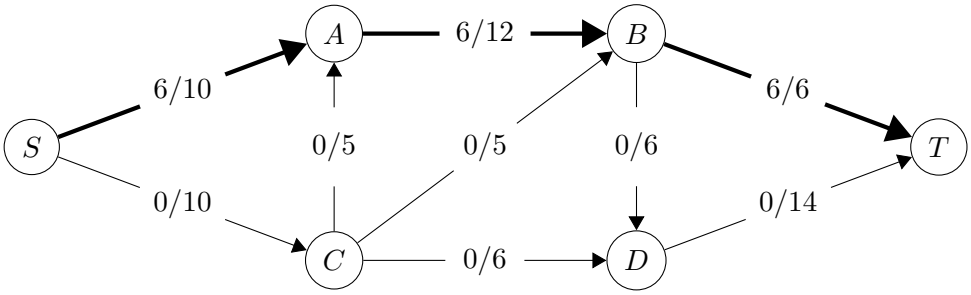
Follow the Ford-Fulkerson Algorithm to compute the max flow of the flow network illustrated below. Show each step to compute the max flow and also show the min cut of the flow network.



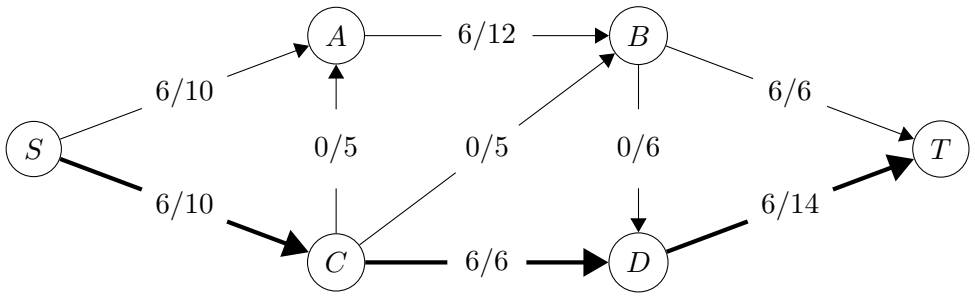
Solution



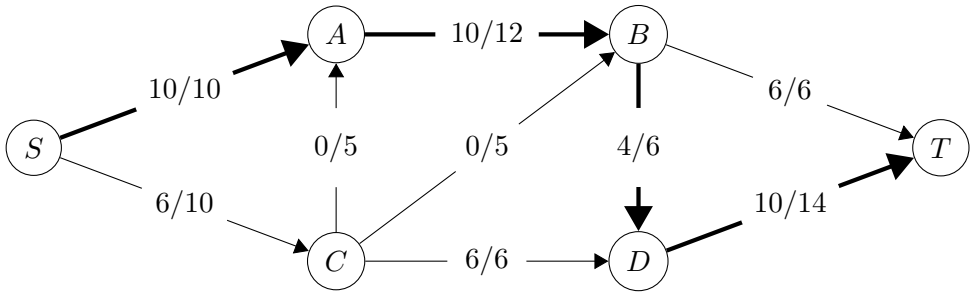
Add 6 to this path.



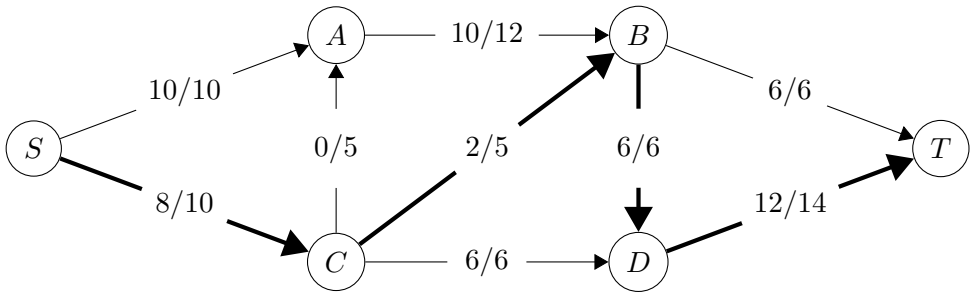
Add 6 to this path.



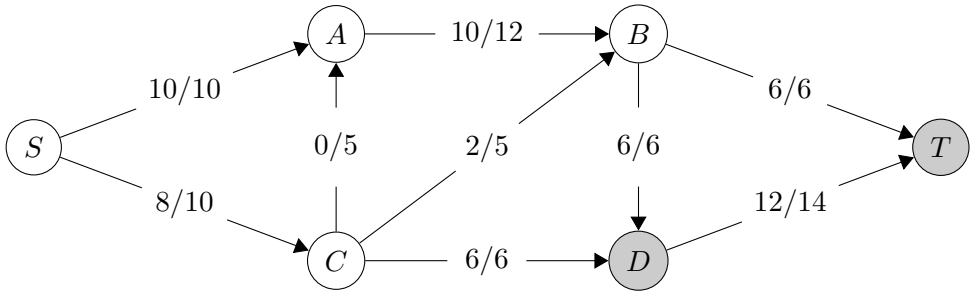
Add 4 to this path.



Add 2 to this path.



$|f| = 18$
Min cut: $S = \{S, A, B, C\}$, $T = \{D, T\}$



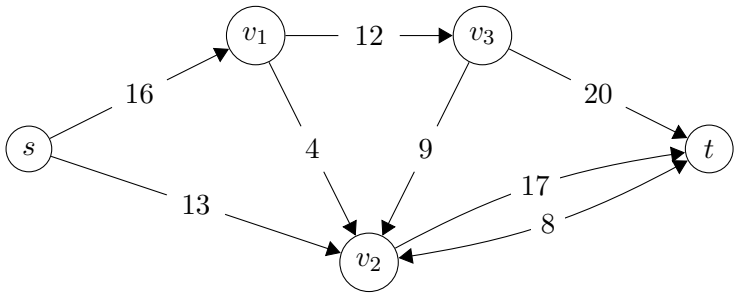
5.8 Max Flow

5.8.1 Old Exam Problems

Spring 2017 #L3

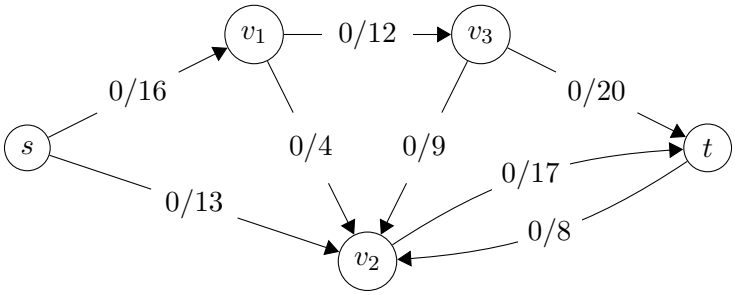
(Same instructions, different graph, as S15 #S7)

The Edmonds-Karp Algorithm (EK) follows the basic Ford-Fulkerson method with breadth-first search to choose the shortest augmenting path (in terms of the number of edges involved) for computing the maximum flow iteratively from vertex s to vertex t in a weighted directed graph. Illustrate the maximum flow computation process (including the augmenting path chosen for each iteration and its resulting residual network) via EK for the graph depicted below.

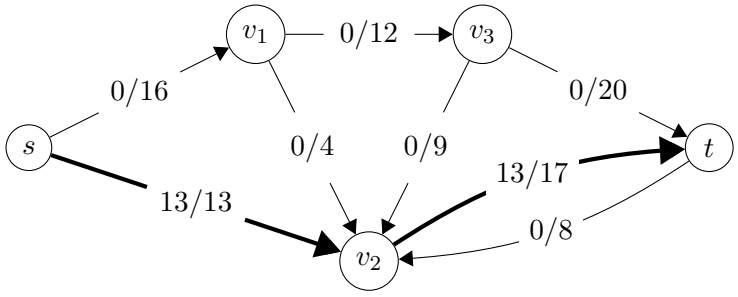


SHOULD THE EDGE WITH WEIGHT 8 BE BIDIRECTIONAL? [I'm going to assume not.]

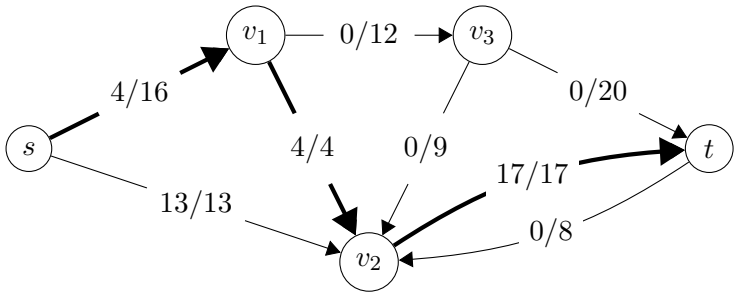
Solution



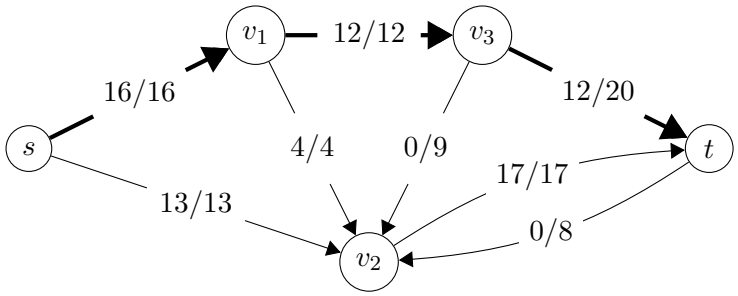
First DFS run.



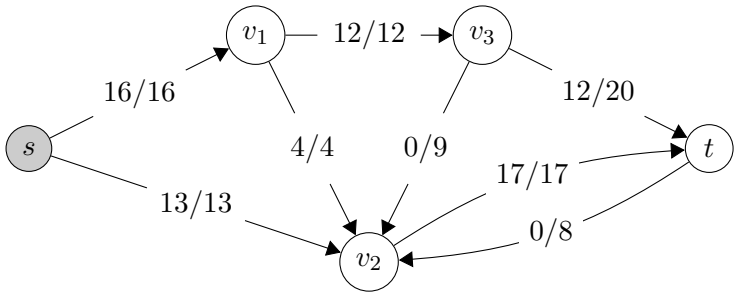
Second run of BFS. Whether you go through $v2$ or $v3$ depends on the order in which the vertices are listed.

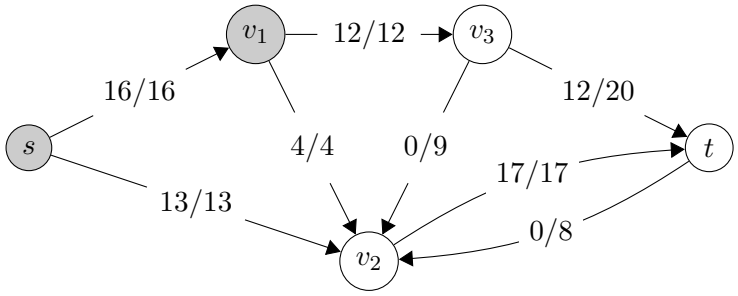


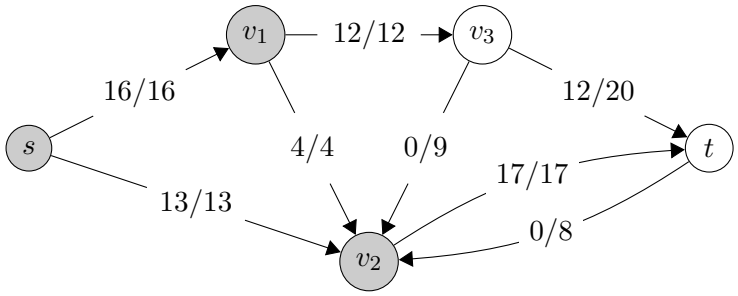
Third run of BFS.



Solution. $|f| = 29$, Min-cuts illustrated.





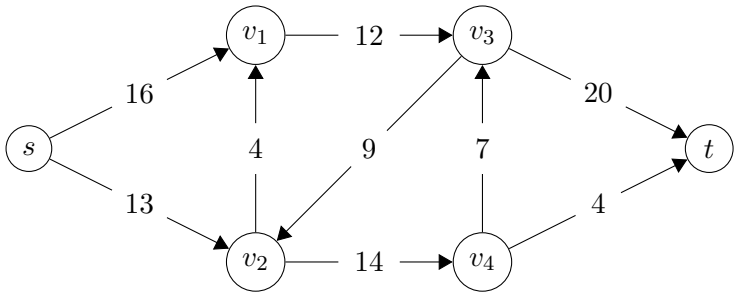


Solution

Spring 2015 #S7

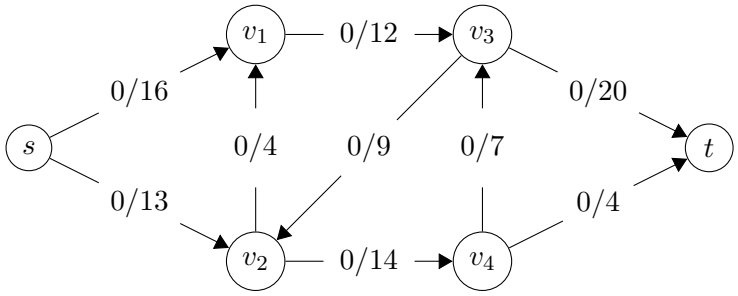
(Same instructions, different graph, as S17 #L3)

The Edmonds-Karp Algorithm (EK) follows the basic Ford-Fulkerson method with breadth-first search to choose the shortest augmenting path (in terms of the number of edges involved) for computing the maximum flow iteratively from vertex s to vertex t in a weighted directed graph. Illustrate the maximum flow computation process (including the augmenting path chosen for each iteration and its resulting residual network) via EK for the graph depicted below.

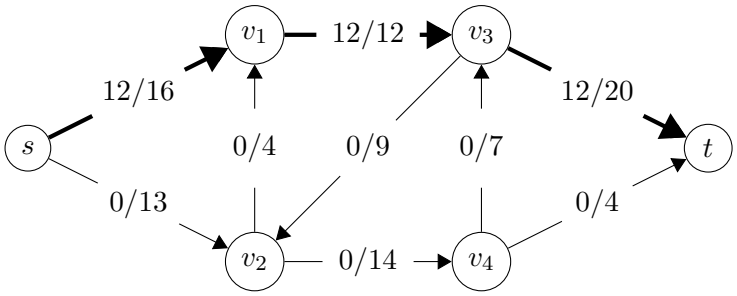


Solution

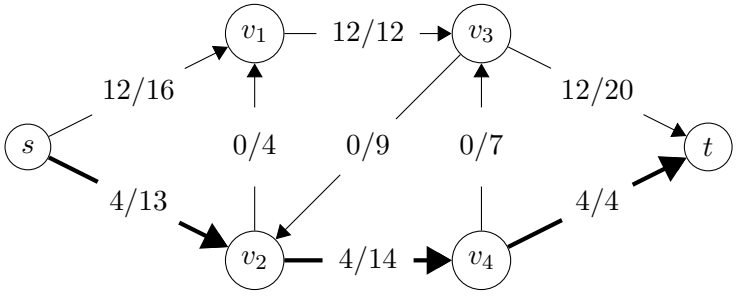
Assume vertices are listed in this order: $\{s, v_1, v_2, v_3, v_4, t\}$



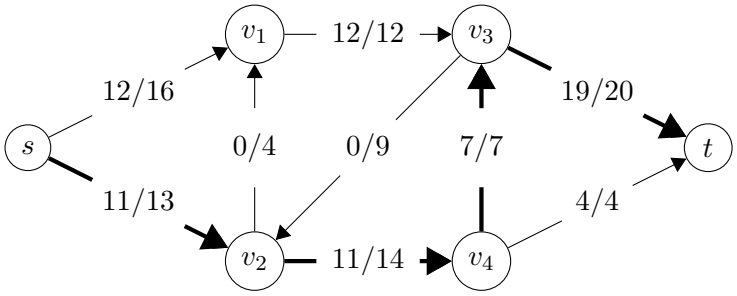
First BFS run.



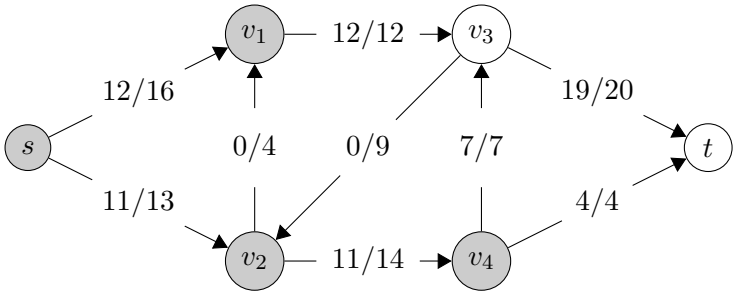
Second BFS run.



Third BFS run.



Solution. $|f| = 23$, min-cut as shown.



Chapter 6

Dynamic Programming

6.1 Knapsack Problem, §16.2, Exercise 35-7

We have n items. The i^{th} item has value v_i and weight w_i. We have a knapsack with total weight capacity W. We need to choose a set of items to take that will maximize the value without exceeding the weight limit.

In the **0-1 knapsack problem**, the values and weights are integers, and we can only choose between taking or not taking an item. We cannot take part of an item nor more than one of an item.

In the **fractional knapsack problem**, we can take fractions of items.

Both problems exhibit optimal substructure. Consider the most valuable load that weighs at most W pounds. If we remove item j from the load, the remaining load must be the most valuable load weighing at most $W - w_j$ pounds that we can take from the $n - 1$ original items excluding j.

The fractional knapsack problem has the greedy-choice property, and thus can be solved with a greedy algorithm. Sort the items by v_i/w_i, value per pound. Take as much as possible of the most valuable item. If it runs out, go to the next most valuable. Continue until the knapsack reaches its weight limit. The greedy algorithm runs in $O(n \lg n)$ time, basically the time to sort.

The 0-1 problem cannot be solved using a greedy algorithm. When we consider whether to include an item in the knapsack, we must compare the solution to the subproblem that includes the item with the solution to the subproblem that excludes the item.

6.1.1 0-1, Top-Down with Memoization

W is the weight capacity of the knapsack.

n is the number of objects.

$v[i]$ is the value of object i.

$w[i]$ is the weight of object i.

W = 6

n = 5

```
w = [0,5,4,3,2,1]
v = [0,1,2,3,4,5]
Memo = [[0 for x in range (W+1)] for y in range (n+1)]
List = [["" for x in range (W+1)] for y in range (n+1)]

for i in range (n+1):
    for j in range (W+1):
        if i==0 or j==0:
            Memo[i][j] = 0
        # If the weight of the object we're considering
        # is less than the knapsack capacity we're considering,
        # then go with the knapsack value for that capacity
            # without that object, i.e.
        # the value in the row above.
        elif w[i]>j:
            Memo[i][j] = Memo[i-1][j]
        else:
            # Given that the current knapsack capacity is j,
                # and the object we're considering has weight w[i],
            # We could fit it into a knapsack with j-w[i] pounds.
            # The current value of the optimal mix for j-w[i] pounds
                # is in Memo[i-1][j-w[i]].
            # If adding object i would give us a better total value at weight j
                # than the current optimal value, Memo[i-1][j],
            # Then add it.

            # Also, keep track of which objects are in the bag.

            Memo[i][j] = Memo[i-1][j]
            List[i][j] = List[i-1][j]
            if Memo[i-1][j-w[i]] + v[i] > Memo[i-1][j]:
                Memo[i][j] = Memo[i-1][j-w[i]] + v[i]
                List[i][j] = List[i-1][j-w[i]] + " " + str(i)
                if List[i][j] != List[i][j-1]:
                    print ("List is now ", List[i][j])

for row in Memo:
    print (row)

for row in List:
```

```
    print (row)

print ()
print ("Max value is ", Memo[n][W], "with objects ", List[n][W])
```

6.1.2 Old Exam Questions

Fall 2015 #L4

a. Describe maximum clique problem.
b. Write a pseudo code to obtain the maximum clique of a graph with N nodes and E edges using generate and test strategy. That is, generate all possible subsets of vertices and test whether the subset is a clique. (Make any assumptions explicitly.)
c. Find the time and space complexity of your algorithm.
d. Describe the 0-1 knapsack problem.
e. Show that the 0-1 knapsack problem belongs to NP-class.
f. Briefly describe a practical way of solving a 0-1 knapsack problem and its time complexity assuming that the knapsack capacity is K and there are M objects. The weight and profit of an object i are denoted by w_i and p_i respectively.

Solution

a. A *clique* is a complete subgraph. The *[maximum] clique problem* is to find a largest complete subgraph in a graph. It is known to be NP-complete.
b. Loop over all sets of vertices, and test whether each is complete. If it is, find its size. If a clique is larger than the previously-found largest clique, write it down.
c. For a graph $G(V, E)$, the number of subgraphs is

$$\sum_{k=0}^{|V|} \binom{|V|}{k} = 2^{|V|}$$

   For each of those subgraphs, we need to test whether it is complete, which takes $O(k^2)$ time for a subgraph of size k. It doesn't really matter, however, for a graph of nontrivial size, because the time complexity is already exponential.
   The size complexity is much more tame, just a small multiple of the size of the data structure storing the graph.
d. In the 0-1 knapsack problem, we have M objects, with each object i having positive integer weight w_i and profit v_i, and a knapsack with weight capacity K. We have to choose whether to include (or not include) each of the M objects in the knapsack in order to maximize the total value while not exceeding the capacity K.
e. To show that a problem belongs to the NP class, we have to show that we can verify a solution in polynomial time. A solution to the 0-1 knapsack problem is a subset of the M objects to

be included in the knapsack, or, equivalently, a binary M-tuple signifying whether each of the M objects is included. To verify the solution is to add up the weights of the included objects, verify that the total weight is less than K, add up the values of the included objects, and verify that the total value is the value in the certificate. This verifying the solution should take linear time, which is a trivial case of polynomial time; therefore, the 0-1 knapsack problem is in class NP.

f. See that the algorithm has optimal substructure.
Let the M objects be ordered, and let $V(n, w)$ be the maximum possible value of a subset of the first n objects having no more than total weight w. We see that $V(0, w) = 0$ and $V(n, 0) = 0$, signifying the cases where there are no objects or no weight capacity.
We have this optimal substructure, that the maximum total value of a subset of the first n objects, $V(n, w)$, is, if the last item is included, $V(n-1, w-w_i) + v_i$, or if the last item is not included, $V(n-1, w)$. The determination on whether to include the last of the n objects depends on which is greater.

$$V(n, w) = \max(V(n-1, w-w_i) + v_i, V(n-1, v))$$

To solve the 0-1 knapsack problem, build up an $(M+1) \times (K+1)$ array with zeros in the first row and column, and build each row according to the recursion above, keeping track of which items are included to give each total value.
The time complexity is $O(MK)$.

Fall 2016 #L2

1. Explain what do you understand by "principle of optimality" in the context of dynamic programming.
2. Characterize 0-1 knapsack problem in terms of objective function, constraints, and the time and space complexity. (Assume there are n objects. Suppose an object i has weight w_i and profit p_i. The overall capacity of the container is W).
3. Show the 0-1 knapsack problem belong to NP-class.
4. Does it belong to P-class? (Provide an explanation accordingly.)
5. Write down the basic rule that satisfy the principle of optimality and domain related constraints to the following problems:
   (a) 0-1 knapsack problem
   (b) Pairwise shortest path problem

Solutions

1. From Bellman, "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision."

A dynamic programming problem builds on optimal substructure and overlapping subproblems. A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems. A problem has overlapping subproblems when a recursive algorithm solves the same subproblems repeatedly. A dynamic programming algorithm will save the results of the subproblems and look up the solutions rather than recalculating.

2. Find a subset A of the ordered set of objects B.

   Objective function Maximize $\sum_{i \in A} p_i$.

   Constraints $\sum_{i \in A} w_i \leq W$

3. (See solution to previous exam.)
4. The 0-1 knapsack problem has a dynamic programming algorithm, but it is only pseudo-polynomial; that is, the algorithm is time complexity is polynomial in the value of the input, but may be exponential in the number of bits required to represent the input; thus, the knapsack problem is weakly NP-complete and does not belong to P-class.
5. For the 0-1 knapsack problem, the rule that satisfies the principal of optimality is this

Spring 2016 #L2

1. Explain what you understand by "principle of optimality."
2. Write down the basic rule that satisfies the principle of optimality and domain related constraints to the following problems.
   (a) Knapsack Problem
   (b) Pairwise Shortest Path Problem
   (c) Chain Matrix Multiplication Problem
3. The optimal solution to the 0-1 knapsack problem belongs to NP-class. John says dynamic program formulation optimally solves the 0-1 knapsack problem. If you agree with John, explain why; otherwise, explain why not.

Spring 2018 #L4

a. Define the following classes of a decision problem: P, NP, and NP-completeness.
b. Consider the 0-1 knapsack problem with n objects each with its respective pre-defined profit. The objective is to maximize the total profit that can be accommodated into a container of capacity W. Define appropriate notations for weight and profit of objects, formulate the problem.
c. Convert of the problem that you have defined in (b) into a decision problem.
d. Show the problem that you have defined in (c) belongs to NP-class.
e. Does the problem in (d) belong to the P-class or NP-completeness. (Justify your answer.)

f. If principle of optimality be applicable to solve the problem defined in (c), formulate it. Otherwise, explain why not.

g. What would be your explanation, if 0-1 knapsack problem is solved by dynamic programming in polynomial time?

Solutions

a.

b.

c.

d.

e.

f.

g. Since the 0-1 knapsack problem is only pseudopolynomial, it is weakly NP-complete. If it can be solved in polynomial time, then P = NP.

Spring 2019 #L2

Knapsack problem: Suppose you want to pack a knapsack with weight limit W. Item i has an integer weight w_i and real value v_i. Your goal is to choose a subset of items with a maximum total value subject to the total weight $\leq W$.

Let $M[n, W]$ denote the maximum value that a set of items $\in \{1, 2, \ldots, n\}$ can have such that the weight is no more than W. We have the following recursive formula.

$$M[n,W] = \begin{cases} 0 & \text{if } n = 0 \text{ or } W = 0 \\ M[n-1,W] & \text{if } w_n > W \\ \max(M[n-1,W-w_n] + v_n, M[n-1,W]) & \text{otherwise} \end{cases}$$

The time complexity of a simple recursive procedure as given below is exponential.

```
M(n,W)
{
        if (n==0 or W==0)
                return 0;
        if (w_n > W)
                result = M(n-1,W);
        else
                result = max{v_n + M(n-1,W-w_n), M(n-1,W)};
        return result;
}
```

Provide a dynamic programming solution of the Knapsack problem after adding two lines of code to the above procedure. (Hint: Use a table to memorize the results.)

Solution

```
M(n, W)
{
        for (i=0; i<=n; i++){
                for (j=0; j<=W; j++){
                        if (i==0 or j==0)
                                M[i,j] = 0;
                        else if (w_n > j)
                                M[i,j] = M[i-1,j];
                        else
                                M[i,j] = max{v_i + M[i-1, j-w_i], M[i-1, j]};
                }
        }
        return M;
}
```

Chapter 7

Greedy Algorithms

7.1 Huffman Codes

Binary tree A structure defined on a finite set of nodes that either

- Contains no nodes, or
- Is composed of three disjoint sets of nodes:
  - A root node,
  - A binary tree called its *left subtree*, and
  - A binary tree caled its *right subtree*.

Full binary tree Each node is either a leaf or has exactly two children.
A full binary tree with n leaf nodes has $n - 1$ internal nodes.

Complete binary tree A full binary tree, all of whose leaves have the same depth. If h is the height of the tree, with a tree with only a root having height 0, a complete binary tree has 2^h leaf nodes and $2^h - 1$ internal nodes.

Binary code or just *code*. Each character is represented by a unique binary string.

Fixed-length code

Variable-length code

Prefix code No codeword is also a prefix of some other codeword. A prefix code can always achieve the optimal data compression.

Greedy Choice Property We can assemble a globally optimal solution by making locally optimal (greedy) choices. We do not consider the results of subproblems.

Optimal Huffman Code Same as *Huffman code.* An optimal prefix code generated by Huffman's greedy algorithm.

- An optimal code for a file is always represented by a full binary tree, because if the tree were not full, we could reduce the length of at least one codeword by making it optimal.

- If C is the alphabet from which the characters are drawn and all character frequencies are positive, then the tree for an optimal prefix code has exactly $|C|$ leaves and (therefore) $|C|-1$ internal nodes.
- The depth of a leaf is the length of its codeword.
- For a character $c \in C$, let $freq(c)$ be its frequency in the file and $d(T, c)$ be its depth in the tree T. The number of bits required to encode the file is

$$B(T) = \sum_{c \in C} freq(c) \cdot d(T, c),$$

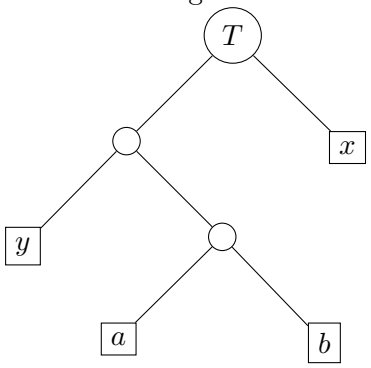
  which we define as the *cost* of the tree T.
- The Huffman Code is a prefix code generated by a greedy algorithm.
  - Make a list, Q, of the characters with their frequencies.
  - Extract two elements with the lowest frequency.
  - Combine them into a tree as the two child nodes, make the frequency of the root the sum of the two frequencies, and put the root in the list, Q.
  - Repeat until the list is empty.

7.1.1 Old Exam Questions

Spring 2018, #S3

(Lemma 16.2, §16.3, page 433) Sketch a proof of the Lemma below, using the tree provided.

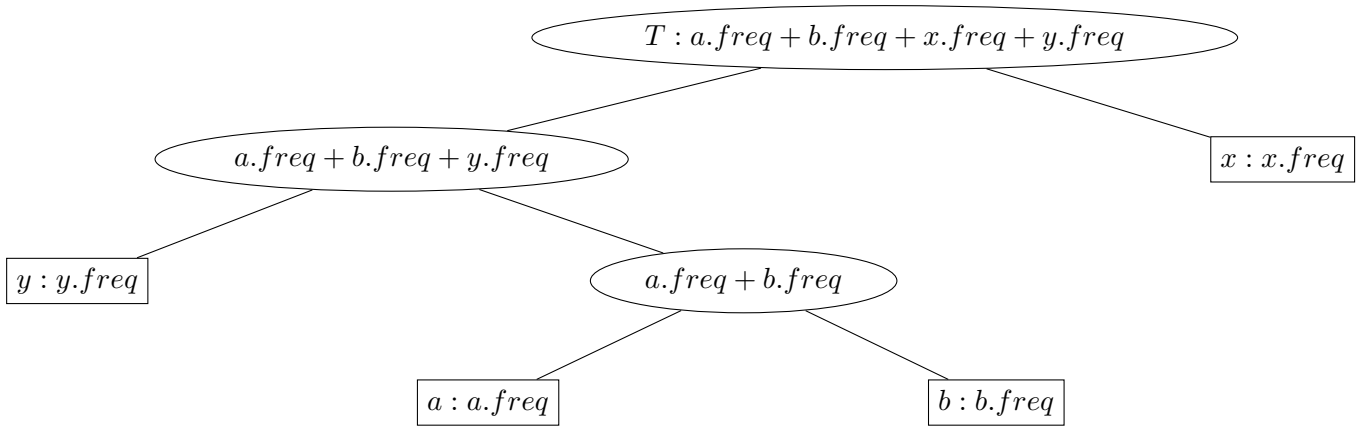
Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.



Solution

[Note: This question would be more reasonable if it stated that it related to Huffman Codes. Are we supposed to be able to prove every lemma in the textbook, out of context?]

This lemma proves that Huffman's algorithm satisfies the greedy-choice property.



For a set of characters C, the total cost (number of bits) of a Huffman tree T, $B(T)$, is the sum of the product of the frequency of each character and its depth in the tree, because the depth gives the number of encoding bits for that character. Let c be a character, $c.freq$ its frequency, and $d_T(c)$ its depth.

$$\begin{aligned} B(T) &= \sum_{c \in C} d_T(c) \cdot c.freq \\ &= 1 \cdot x.freq + 2 \cdot y.freq + 3 \cdot a.freq + 3 \cdot b.freq \end{aligned}$$

We will prove the lemma by:

1. Assuming that the given tree is optimal; that is, it has the lowest possible cost.
2. Showing that, without increasing the cost, we can reorganize the tree such that x and y are the lowest leaf nodes, giving them codewords that have the same length and differ only in the last bit.

In the above diagram, we are given that x and y have the lowest frequencies, so $x.freq \leq \min(a.freq, b.freq)$ and $y.freq \leq \min(a.freq, b.freq)$.

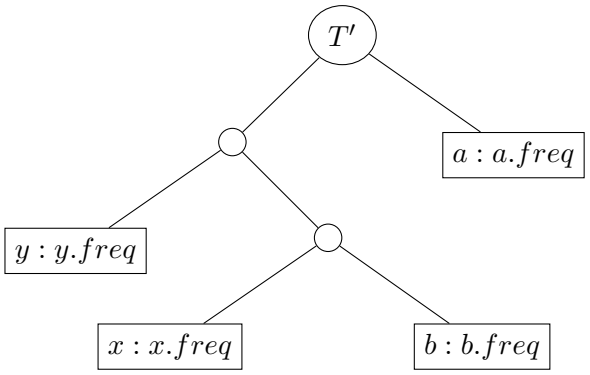
From the tree structure, we have $a.freq \leq b.freq$.

Without loss of generality, we will assume that $x.freq \leq y.freq$.

Putting all of that together, we have

$$x.freq \leq y.freq \leq a.freq \leq b.freq$$

Create tree T' by swapping x and a.

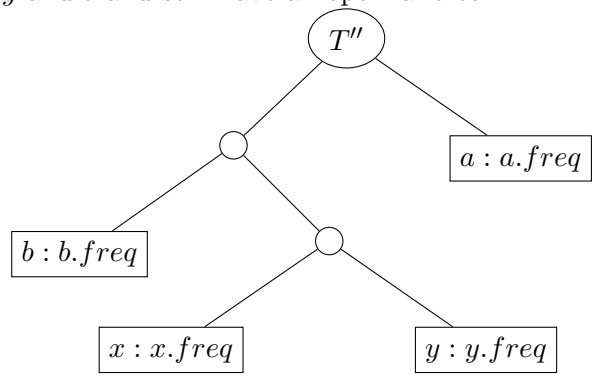


The cost of this tree is $B(T') = 1 \cdot a.freq + 2 \cdot y.freq + 3 \cdot x.freq + 3 \cdot b.freq$. We want to show that the cost of T' is not more than the cost of T.

$$\begin{aligned} B(T) - B(T') &= (1 \cdot x.f + 2 \cdot y.f + 3 \cdot a.f + 3 \cdot b.f) - (1 \cdot a.f + 2 \cdot y.f + 3 \cdot x.f + 3 \cdot b.f) \\ &= (1 \cdot x.f + 3 \cdot a.f) - (1 \cdot a.f + 3 \cdot x.f) \\ &= 2(a.freq - x.freq) \\ &\geq 0 \\ B(T) &\geq B(T') \end{aligned}$$

Because we assumed T was optimal, we also have $B(T) \leq B(T')$; therefore, $B(T') = B(T)$. We have swapped x and a without changing the cost of the tree, so T' is also optimal.

Similarly, we can swap y and b and still have an optimal tree.



$$\begin{aligned} B(T') - B(T'') &= (1 \cdot a.f + 2 \cdot y.f + 3 \cdot x.f + 3 \cdot b.f) - (1 \cdot a.f + 2 \cdot b.f + 3 \cdot x.f + 3 \cdot y.f) \\ &= (2 \cdot y.f + 3 \cdot b.f) - (3 \cdot y.f + 2 \cdot b.f) \\ &= 2(b.freq - y.freq) \\ &\geq 0 \\ B(T') &\geq B(T'') \end{aligned}$$

But since T' is optimal, $B(T') \leq B(T'')$, so $B(T') = B(T'')$ and $B(T) = B(T'')$.

Thus, we have created an optimal prefix code where x and y differ only by their last digit.

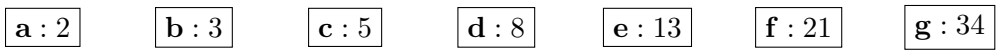
The implication of this lemma is that Huffman's algorithm has the greedy choice property, because building up an optimal tree can begin with mergers of the nodes of lowest frequency.

Spring 2015, #S6

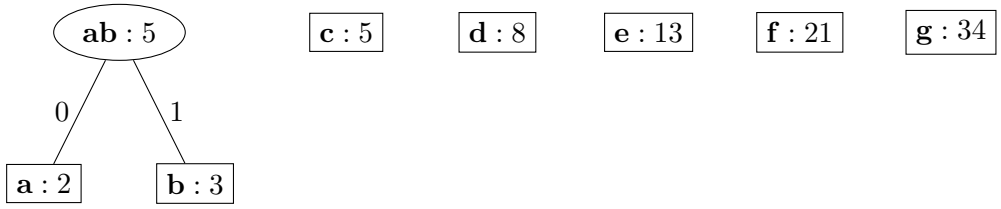
Show your construction of an optimal Huffman code for the set of 7 frequencies:

a : 2 **b** : 3 **c** : 5 **d** : 8 **e** : 13 **f** : 21 **g** : 34

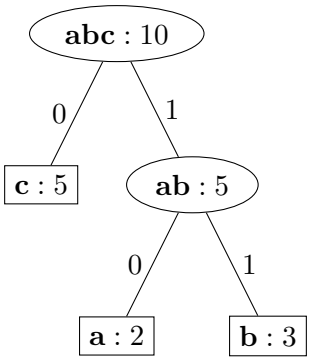
Solution



$$Q = \{a : 2, b : 3, c : 5, d : 8, e : 13, f : 21, g : 34\}$$

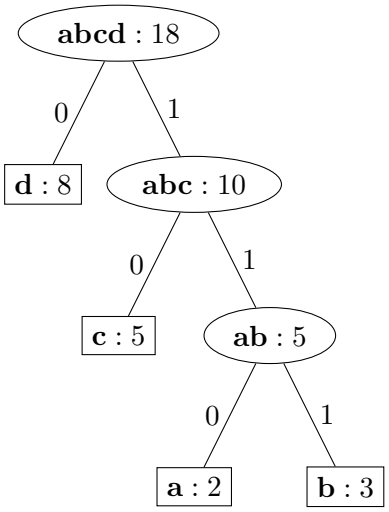


$$Q = \{c : 5, d : 8, e : 13, f : 21, g : 34, ab : 5\}$$



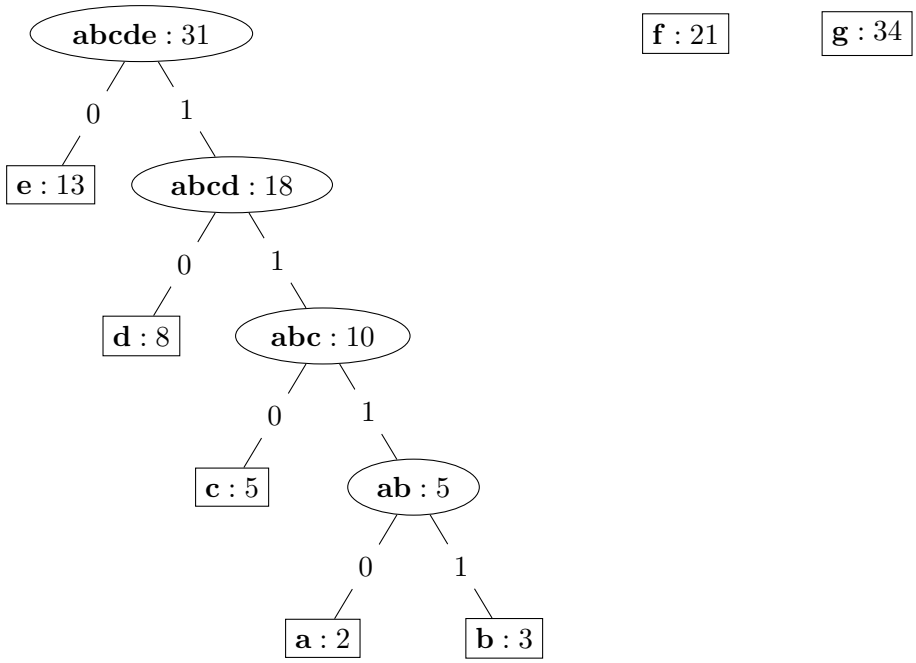
d : 8 **e** : 13 **f** : 21 **g** : 34

$$Q = \{d : 8, e : 13, f : 21, g : 34, abc : 10\}$$

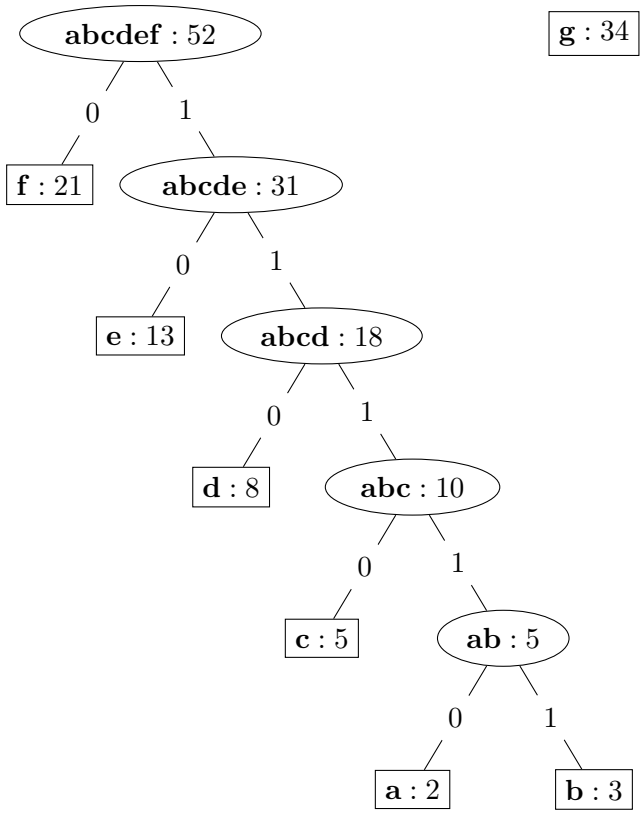


e : 13 **f** : 21 **g** : 34

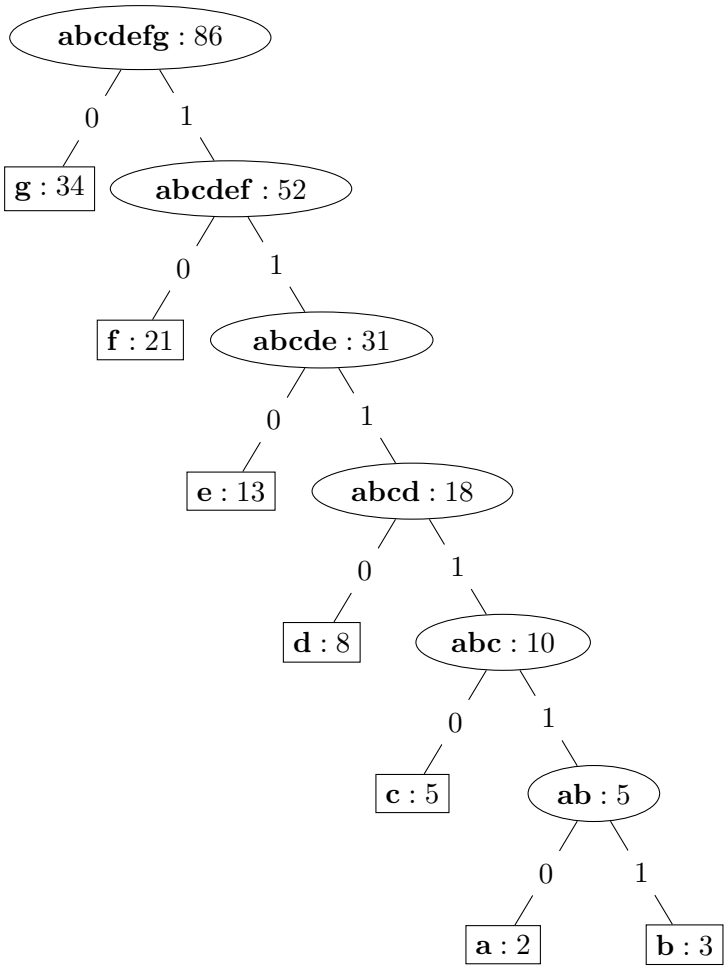
$$Q = \{e : 13, f : 21, g : 34, abcd : 18\}$$



$$Q = \{f : 21, g : 34, abcde : 31\}$$



$$Q = \{g : 34, abcdef : 52\}$$



$Q = \{abcdefg : 86\}$

Prefix Codes

a	`111110`
b	`111111`
c	`11110`
d	`1110`
e	`110`
f	`10`
g	`0`

Spring 2019 #S3

Show your construction of an optimal Huffman code for the set of 10 frequencies:

a:2 **b**:6 **c**:5 **d**:8 **e**:13 **f**:34 **g**:34 **h**:15 **i**:27 **j**:9

Solution

$Q_0 = \{\underline{\mathbf{a} : 2},\ \mathbf{b} : 6,\ \underline{\mathbf{c} : 5},\ \mathbf{d} : 8,\ \mathbf{e} : 13,\ \mathbf{f} : 34,\ \mathbf{g} : 34,\ \mathbf{h} : 15,\ \mathbf{i} : 27,\ \mathbf{j} : 9\}$

$Q_1 = \{\underline{\mathbf{b} : 6},\ \mathbf{d} : 8,\ \mathbf{e} : 13,\ \mathbf{f} : 34,\ \mathbf{g} : 34,\ \mathbf{h} : 15,\ \mathbf{i} : 27,\ \mathbf{j} : 9,\ \underline{\mathbf{ac} : 7}\}$

$Q_2 = \{\underline{\mathbf{d} : 8},\ \mathbf{e} : 13,\ \mathbf{f} : 34,\ \mathbf{g} : 34,\ \mathbf{h} : 15,\ \mathbf{i} : 27,\ \underline{\mathbf{j} : 9},\ \mathbf{bac} : 13\}$

$Q_3 = \{\underline{\mathbf{e} : 13},\ \mathbf{f} : 34,\ \mathbf{g} : 34,\ \mathbf{h} : 15,\ \mathbf{i} : 27,\ \underline{\mathbf{bac} : 13},\ \mathbf{dj} : 17\}$

$Q_4 = \{\mathbf{f} : 34,\ \mathbf{g} : 34,\ \underline{\mathbf{h} : 15},\ \mathbf{i} : 27,\ \underline{\mathbf{dj} : 17},\ \mathbf{ebac} : 26\}$

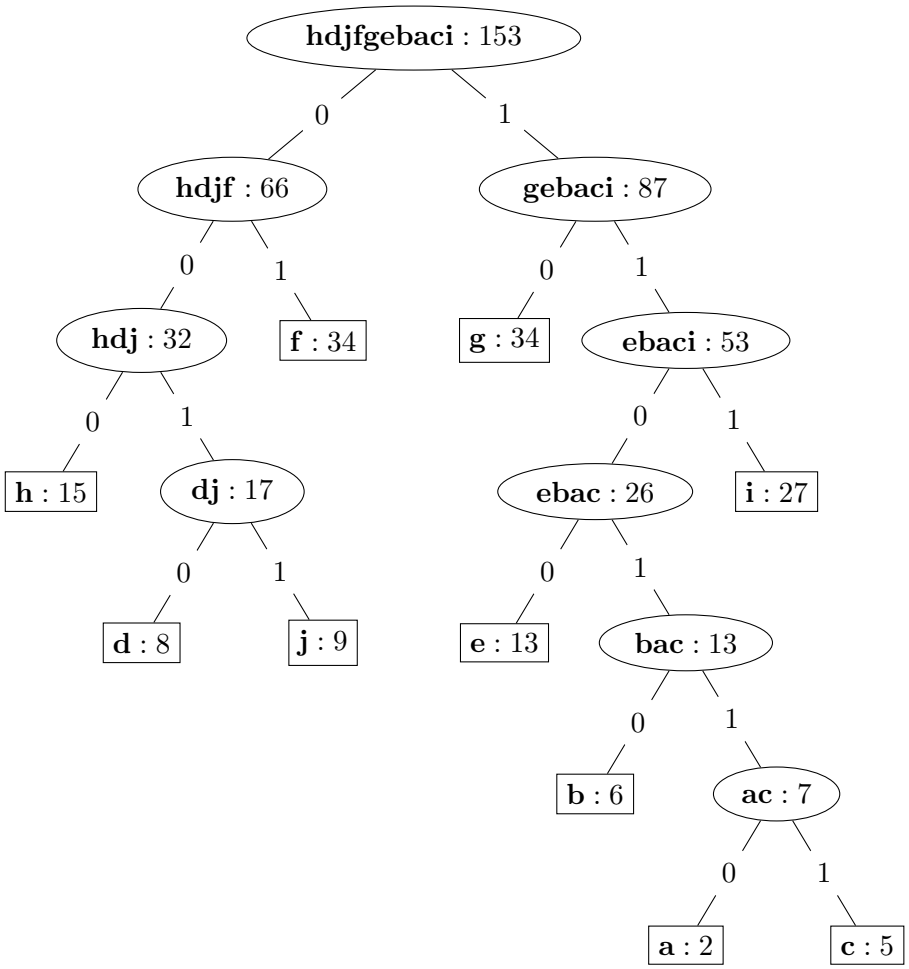
$Q_5 = \{\mathbf{f} : 34,\ \mathbf{g} : 34,\ \underline{\mathbf{i} : 27},\ \underline{\mathbf{ebac} : 26},\ \mathbf{hdj} : 32\}$

$Q_6 = \{\underline{\mathbf{f} : 34},\ \mathbf{g} : 34,\ \underline{\mathbf{hdj} : 32},\ \mathbf{ebaci} : 53\}$

$Q_7 = \{\underline{\mathbf{g} : 34},\ \underline{\mathbf{ebaci} : 53},\ \mathbf{hdjf} : 66\}$

$Q_8 = \{\underline{\mathbf{hdjf} : 66},\ \underline{\mathbf{gebaci} : 87}\}$

$Q_9 = \{\mathbf{hdjfgebaci} : 153\}$



Prefix Codes

a 110110
b 11010
c 110111
d 0010
e 1100
f 01
g 10
h 000
i 111
j 0011

Fall 2018 #S6

Show your construction of an optimal Huffman code for the set of 7 frequencies:
$\mathbf{a} : 3, \mathbf{b} : 12, \mathbf{c} : 5, \mathbf{d} : 20, \mathbf{e} : 16, \mathbf{f} : 34, \mathbf{g} : 18$

7.2 Matrix Chain Multiplication

Matrix Chain Multiplication is almost identical to optimal binary search trees.

Multiplying two matrices of dimensions $a \times b$ and $b \times c$ requires abc number of multiplications (plus lots of addition), so it's an $O(abc)$ operation.

$$\underset{a\times b}{\mathrm{A}} \times \underset{b\times c}{\mathrm{B}} = \underset{a\times c}{\mathrm{C}}$$

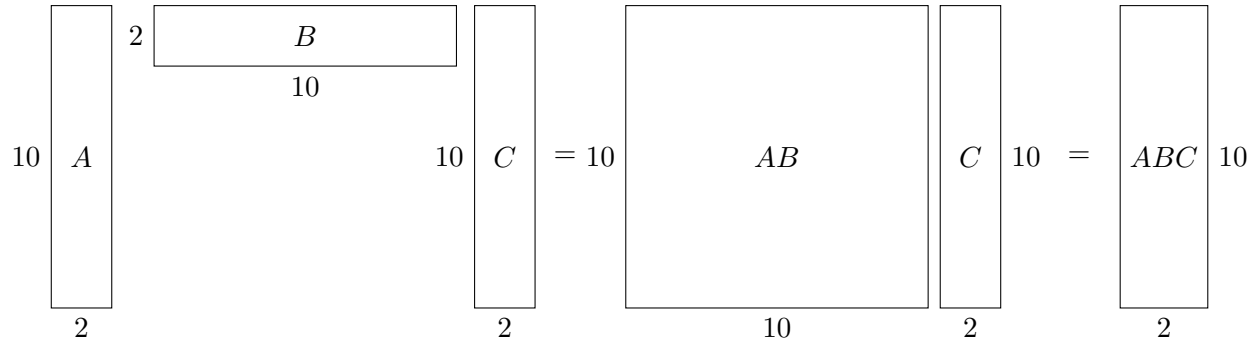
When multiplying three matrices, since matrix multiplication is associative (but not commutative), we have two options of how to multiply.

$$\underset{a\times b}{\mathrm{A}} \times \underset{b\times c}{\mathrm{B}} \times \underset{c\times d}{\mathrm{C}} = \underbrace{\left(\underset{a\times b}{\mathrm{A}} \times \underset{b\times c}{\mathrm{B}}\right)}_{O(abc)} \times \underset{c\times d}{\mathrm{C}} = \underset{a\times c}{\mathrm{AB}} \times \underset{c\times d}{\mathrm{C}} \qquad abc + acd = (ac)(b+d)$$
$$\underset{O(acd)}{}$$

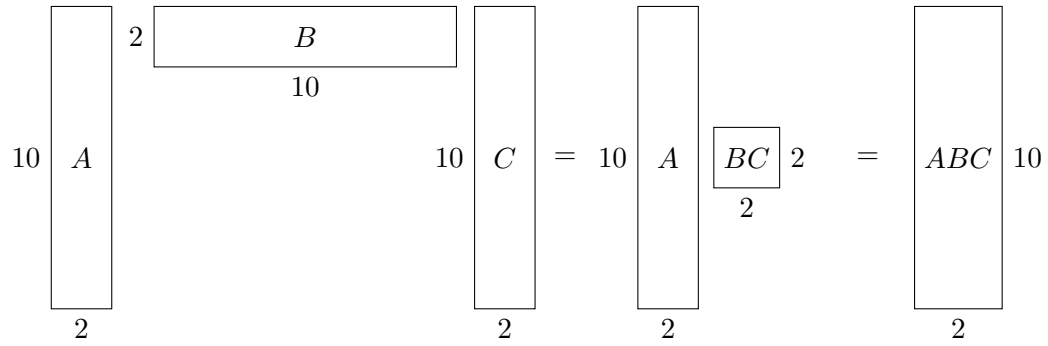
$$\underset{a\times b}{\mathrm{A}} \times \underset{b\times c}{\mathrm{B}} \times \underset{c\times d}{\mathrm{C}} = \underset{a\times b}{\mathrm{A}} \times \underbrace{\left(\underset{b\times c}{\mathrm{B}} \times \underset{c\times d}{\mathrm{C}}\right)}_{O(bcd)} = \underset{a\times b}{\mathrm{A}} \times \underset{b\times d}{\mathrm{BC}} \qquad bcd + abd = (bd)(a+c)$$
$$\underset{O(abd)}{}$$

If for instance, A were 10×2, B were 2×10, and C were 10×2, here's an illustration of the difference.

$ABC = (AB)C$: $(10 \cdot 10)(2 + 2) = 400$ multiplications.



$ABC = A(BC)$: $(2 \cdot 2)(10 + 10) = 80$ multiplications.



7.2.1 Exercise 15.2-3

Use the substitution method to show that the solution for the recurrence (15.6) is $\Omega(2^n)$.

Let $P(n)$ be the number of ways to parenthesize a sequence of n matrices.

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \end{cases}$$

$$\begin{aligned}
P(1) &= 1 && 2^1 = 2 \\
P(2) &= P(1)P(1) = 1 && 2^2 = 4 \\
P(3) &= P(1)P(2) + P(2)P(1) = 1 \cdot 1 + 1 \cdot 1 = 2 && 2^3 = 8 \\
P(4) &= P(1)P(3) + P(2)P(2) + P(3)P(1) = 1 \cdot 2 + 1 \cdot 1 + 2 \cdot 1 = 5 && 2^4 = 16 \\
P(5) &= P(1)P(4) + P(2)P(3) + P(3)P(2) + P(4)P(1) = 5 + 2 + 2 + 5 = 14 && 2^5 = 32 \\
P(6) &= 2 \cdot P(1)P(5) + 2 \cdot P(2)P(4) + P(3)P(3) = 2 \cdot 14 + 2 \cdot 5 + 4 = 42 && 2^6 = 64 \\
P(7) &= 2 \cdot P(1)P(6) + 2 \cdot P(2)P(5) + 2 \cdot P(3)P(4) = 132 && 2^7 = 128 \\
P(8) &= 2 \cdot P(1)P(7) + 2 \cdot P(2)P(6) + 2 \cdot P(3)P(5) + P(4)P(4) = 429 && 2^8 = 256
\end{aligned}$$

We have $2^n < P(n)$ for $n = 7$. Prove that, for all $k \geq 7$, if $2^k < P(k)$, then $2^{k+1} < P(k+1)$.

$$\begin{aligned}
2^k &< P(k) \\
2 \cdot 2^k &< 2 \cdot P(k) \\
&= 2 \cdot P(1)P(k) \\
&= P(1)P(k) + P(k)P(1) \\
2 \cdot 2^k &< P(1)P(k) + P(2)P(k-1) + \cdots + P(k-1)P(2) + P(k)P(1) \\
2^{k+1} &< P(k+1)
\end{aligned}$$

Therefore, $2^n < P(n)$ for all $n \geq 7$, so 2^n is a lower asymptotic bound on $P(n)$, so $P(n) = \Omega(2^n)$.

We don't know exactly how computationally expensive it would be to exhaustively search all options for parenthesizing a product of n matrices, but it's at least exponentially expensive, so we should try to find another way.

7.2.2 Optimal Substructure

For any i and j such that $1 \leq i < j \leq n$,

$$A_1 \times A_2 \times \cdots \times A_i \times A_{i+1} \times \cdots \times A_j \times \cdots \times A_n$$

there is a k such that $i \leq k < j$ where the parenthesization breaks,

$$A_i \times \cdots \times A_k \times A_{k+1} \times \cdots A_j = (A_i \times \cdots \times A_k) \times (A_{k+1} \times \cdots A_j)$$

The cost of this multiplication is the cost of the multiplication $A_i \times \cdots \times A_k$, plus the cost of the multiplication $A_{k+1} \times \cdots \times A_j$, plus the cost of multiplying the two product matrices.

If the product $A_i \times \cdots \times A_j$ is optimally parenthesized, then both $A_i \times \cdots \times A_k$ and $A_{k+1} \times \cdots \times A_j$ are optimally parenthesized. Proof: If either subsequences's parenthesization could be improved, then improving it would improve the parenthesization of the larger sequence would improve, but we've assumed that it's optimal.

7.2.3 Python Code

This code is from the text and incorporates the example on page 376.

```
def Print_Optimal_Parens (s, i, j):
```

```
    if i==j:
        print ("A ", end='')
        print (i, end=' ')
    else:
        print ("( ", end='')
        Print_Optimal_Parens( s, i, s[i][j] )
        Print_Optimal_Parens( s, s[i][j]+1, j )
        print(") ", end='')


p = [30,35,15,5,10,20,25]
n = len(p) - 1

m = [[0 for x in range (n+1)] for y in range (n+1)]
s = [[0 for x in range (n+1)] for y in range (n)]

for i in range (1, n+1):
    m[i][i] = 0 # Redundant, but who cares?
for l in range (2, n+1):
    for i in range (1, n-l+2):
        j = i+l-1
        m[i][j] = float ("inf")
        for k in range (i, j):
            q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j]
            if q < m[i][j]:
                m[i][j] = q
                s[i][j] = k

for i in range (1,n+1):
    for j in range (1,n+1):
        if m[i][j]>0:
            print ("%5d " % (m[i][j]), end='')
        else:
            print ("       ", end="")
    print ()
print ()

for i in range (1,n):
    for j in range (2,n+1):
        if s[i][j]>0:
```

```
            print ("%2d " % (s[i][j]), end='')
        else:
            print ("    ", end="")
    print ()
print ()

Print_Optimal_Parens(s, 1, 6)
print ()
```

Output

```
  15750  7875  9375 11875 15125
         2625  4375  7125 10500
               750  2500  5375
                    1000  3500
                          5000


 1  1  3  3  3
    2  3  3  3
       3  3  3
          4  5
             5

( ( A_1 ( A_2 A_3 ) ) ( ( A_4 A_5 ) A_6 ) )
```

7.2.4 Textbook Example (page 376)

The textbook gives this example of multiplying six matrices.

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimension	30×35	35×15	15×5	5×10	10×20	20×25

Note that all of the dimensions are multiples of 5. Especially if we're doing this by hand on an exam, or even punching it into a calculator, we can make things go much faster with less chance of error if we divide all of the dimensions by 5. The resulting parenthesization will be the same, the matrix s will be the same, and to get from the number of multiplication operations in m with the simplification to the original, multiply by $5^3 = 125$.

Here's the problem we'll work with, and the corresponding solutions.

Matrix	A_1	A_2	A_3	A_4	A_5	A_6
Dimension	6×7	7×3	3×1	1×2	2×4	4×5

```
p   [6, 7, 3, 1, 2, 4, 5]

m         1     2     3     4     5     6
    1     0   126    63    75    95   121
    2           0    21    35    57    84
    3                 0     6    20    43
    4                       0     8    28
    5                             0    40
    6                                   0

s   2  3  4  5  6
 1  1  1  3  3  3
 2     2  3  3  3
 3        3  3  3
 4           4  5
 5              5

( ( A_1 ( A_2 A_3 ) ) ( ( A_4 A_5 ) A_6 ) )
```

The value $m[i, j]$ indicates, "If we multiply $A_i \times A_{i+1} \times \cdots \times A_j$ in the optimal order, that matrix multiplication requires $m[i, j]$ scalar multiplications." The value $s[i, j]$ indicates, "If we are multiplying $A_i \times A_{i+1} \times \cdots \times A_j$ in the optimal order, then we split the parentheses after $A_{s[i,j]}$.

For instance, when multiplying $\underset{6\times7}{A_1} \times \underset{7\times3}{A_2}$ we have $6\times7\times3 = 126 = m[1, 2]$ scalar multiplications, and if we multiply $\underset{7\times3}{A_2} \times \underset{3\times1}{A_3}$, we have $7 \times 3 \times 1 = 21 = m[2, 3]$ scalar multiplications.

If we want to multiply $\underset{6\times7}{A_1} \times \underset{7\times3}{A_2} \times \underset{3\times1}{A_3}$, we have two options, $\underset{6\times7}{A_1} \times \left(\underset{7\times3}{A_2} \times \underset{3\times1}{A_3} \right)$, and $\left(\underset{6\times7}{A_1} \times \underset{7\times3}{A_2} \right) \times \underset{3\times1}{A_3}$. If the first were optimal, then $s[1, 3]$ would be 1, indicating that the parentheses split after A_1. If the second were optimal, then $s[1, 3]$ would be 2, indicating that the parentheses split after A_2.

The first option, splitting after A_1, requires $7 \times 3 \times 1 = m[2, 3] = 21$ scalar multiplications for the first product, then $6 \times 7 \times 1 = p_0 \cdot p_1 \cdot p_3 = 42$, for a total of 63.

The second option, splitting after A_2, requires $6 \times 7 \times 3 = 126 = m[1, 2]$, then $6 \times 3 \times 1 = p_0 \cdot p_2 \cdot p_3 = 18$, for a total of 144.

We get this from the algorithm by:

$$m[1, 3] = \min \begin{cases} m[1, 1] + m[2, 3] + p_0 \cdot p_1 \cdot p_3 = 0 + 21 + 42 = 63 \\ m[1, 2] + m[3, 3] + p_0 \cdot p_2 \cdot p_3 = 126 + 0 + 18 = 144 \end{cases}$$

If the first option is the min, then $s[1, 3] = 1$. If the other, then $s[1, 3] = 2$.

Continue in like fashion down the diagonal to find $m[2,4]$, s[2,4], $m[3,5]$, s[3,5], $m[4,6], and s[4,6]$.

When multiplying $\underset{6\times7}{A_1} \times \underset{7\times3}{A_2} \times \underset{3\times1}{A_3} \times \underset{1\times2}{A_4}$, there are three ways to split it, after A_1, after A_2, and after A_3.

If we split it after A_1, then we use the previously-calculated optimal way to calculate $A_2 \times A_3 \times A_4$, giving $m[1,1] + m[2,4] + p_0 \cdot p_1 \cdot p_4 = 0 + 35 + 84 = 119$ operations.

If we split it after A_2, then we use the previously-calculated number of operations for $A_1 \times A_2$ and $A_3 \times A_4$ to give $m[1,2] + m[3,4] + p_0 \cdot p_2 \cdot p_4 = 126 + 6 + 36 = 168$ operations .

If we split it after A_3, then we use the previously-calculated optimal way to calculate $A_1 \times A_2 \times A_3$, giving $m[1,3] + m[4,4] + p_0 \times p_3 \times p_4 = 63 + 0 + 12 = 75$ operations.

Since the last option gives the min, $m[1,4] = 75$ and $s[1,4] = 3$.

Continue in like fashion.

To read the parenthesization from s, start with $s[1,n] = s[1,6] = 3$, which tell us to split first after A_3.

$$(A_1 \times A_2 \times A_3)\,(A_4 \times A_5 \times A_6)$$

Then $s[1,3] = 1$ tells us to split $A_1 \times A_2 \times A_3$ after A_1.

$$(A_1\,(A_2 \times A_3))\,(A_4 \times A_5 \times A_6)$$

Then $s[4,6] = 5$ tells us to split $A_4 \times A_5 \times A_6$ after A_5.

$$(A_1\,(A_2 \times A_3))\,((A_4 \times A_5)\,A_6)$$

Voila!

Chapter 8

Hashing

8.1 Hashing

8.1.1 Definitions

A hash is a function h that takes the key k to its hash table position $h(k)$. The universe of keys is large, and the table is much smaller, so $h : U \to T$ is not invertable. Collisions, where $h(k_i) = h(k_j)$, are possible, so a hash method must have a method of resolving collisions.

With chaining, each slot in the hash table is actually a linked list of the elements of U hashed to that table position. The assumption of *simple uniform hashing* is that any given element is equally likely to hash into any of the m table slots, independent of where any other element has hashed to. Let $m = |T|$ and n is the number of elements to be put in the table. The expected value of the number of elements in linked list T_i is n/m. In a hash table in which collisions are solved by chaining, an unsuccessful search takes average-case time $\Theta(1 + n/m)$ under the assumption of simple uniform hashing.

8.1.2 Examples of Hash Functions

Division $h(k) = k \pmod m$. Choose m prime not close to a power of 2.

Multiplication $h(k) = \lfloor m \cdot (\text{fractional part of } kA) \rfloor$ where $A \in (0, 1)$ and $m = |T|$.

Method

- $m = 2^p$
- $w =$ word size (bits), assume $k < 2^w$.
- $A = s/2^w$, $0 < s < 2^w$
- Multiply k by $s = A \cdot 2^w$.
- $ks = r_1 2^w + r_0$
- Extract the p most significant digits of ks to be $h(k)$.

We restrict A to be a fraction of the form $s/2^w$, where w is the word size.

8.1.3 Collision Resolution

Chaining Each slot in the hash table is actually a linked list.

8.1.4 Cuckoo Hashing: NOT IN TEXTBOOK

Cuckoo hashing is a scheme in computer programming for resolving hash collisions of values of hash functions in a table, with worst-case constant lookup time. The name derives from the behavior of some species of cuckoo, where the cuckoo chick pushes the other eggs or young out of the nest when it hatches; analogously, inserting a new key into a cuckoo hashing table may push an older key to a different location in the table. [Wikipedia]

Have two hash tables, each with a different hashing function. When hashing a key k, if $h_1(k)$ is empty in the first table, put it there. If another key, k_1, is there, push it into the second table in $h_2(k_1)$. and put k in $h_1(k)$. If $h_2(k_1)$ is full with k_3, push k_3 into the first table in $h_1(k_3)$. Continue until there's an empty cell.

8.1.5 Perfect Hashing

Perfect hashing is for situations where the keys are static, as in the reserved terms in a programming language or the TOC of a DVD.

Instead of making a linked list for keys hashing to slot j, make a secondary hash table S_j with an associated hash function h_j. The size m_j of the secondary hash table has to be the square of the number n_j of keys hashing to slot j.

8.1.6 Old Exam Questions

Spring 2019 #S2

The hash table is a widely adopted data structure. Explain briefly how perfect hashing works. Separately, what is the situation when a new key cannot be inserted in a Cuckoo hash table separately?

Solution

Perfect hashing: Let m be the size of the hash table T, and m_j be the number of keys hashed to element j of the hash table. For each element j of the table, instead of having a linked list to store the m_j elements, have another hash table of size m_j^2 to hash them with hash function h_j. It is possible to create each of the hash functions h_j so that there are no collisions within that table.

A cuckoo hash table is actually two hash tables, T_1 and T_2 with hash functions h_1 and h_2. When hashing a key k, if $h_1(k)$ is full, put k there, and hash the pushed-out key into the other table. If that slot is full, continue pushing keys between the tables until there is an empty slot.

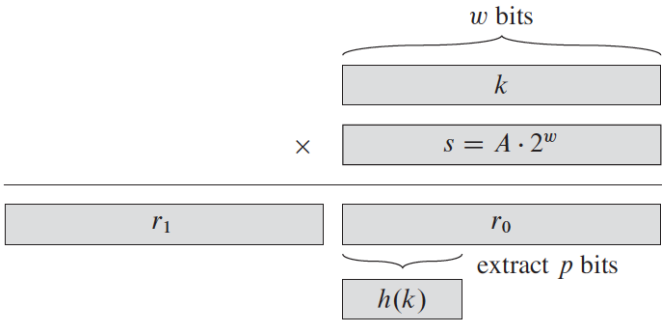
Fall 2018 #S5

The utilization efficiency of a hash table depends heavily on its hashing function(s) employed. Describe with a diagram to illustrate how a multiplication method of hashing functions works on a machine with the word size of w bits for a hash table with 2^p entries, $p < w$.

Solution

This problem refers to Figure 11.4 on page 264.

1. Let k be the key to be hashed, and $h(k)$ be the hash table value in which we will place k. Assume $k < 2^w$.
2. Choose $A \in (0, 1)$, by which we really mean choose $s = A \cdot 2^w$.
3. Multiply sk to get a value of $2w$ bits, $r_1 \cdot 2^w + r_0$.
4. Take the p higher-order bits of r_0 to be $h(k)$.



Spring 2018 #S2

Given that for an open-address hash table with load factor $\alpha = \frac{n}{m} < 1$, the expected number of probes in unsuccessful search under uniform hashing is at most $\frac{1}{1-\alpha}$, how do you prove the expected number of probes in a successful probe under uniform hashing being at most $\frac{1}{\alpha} \ln \left(\frac{1}{1-\alpha} \right)$? (Just give a proof sketch, explaining how many probes are needed to locate existing keys.)

Solution

This one requires summing probabilities. I'm going to pass on this one.

Fall 2016 #S9

Given two hash functions h_1 and h_2 for Cuckoo hashing under two tables, T_1 and T_2,

- Describe the steps involved in inserting a record with the key of k_{new}.

- Cuckoo hashing can be analyzed by the Cuckoo graph, whose nodes denote table entries and links connect pairs of nodes where given keys can be held. State when a new key can be inserted successfully based on the Cuckoo graph.

Solution

- (See above, Spring 2019 #S2.b)
- A new item can be successfully inserted when the Cuckoo graph is acyclic.

Fall 2015 #L1

The utilization efficiency of a hash table depends on its hashing function(s) employed. Describe with a diagram to illustrate how a multiplication method of hashing works on a machine with word size of w bits for a hash table with 2^p entries, $p < w$. Explain briefly how Cuckoo hashing works under two hash functions of h_1 and h_2.

[Solutions above.]

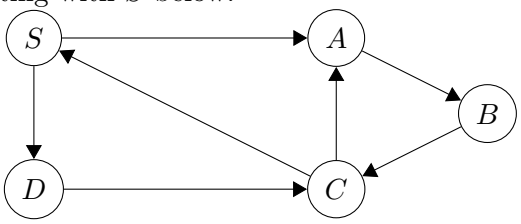
Chapter 9

Old Exams

9.1 Spring 2019

9.1.1 Short Questions (Answer all six questions.)

1. Give a big-O (upper bound) estimate for $f(n) = n\log(n!) + 3n^2 + 2n - 10000$, where n is a positive integer.
2. The hash table is a widely adopted data structure. Explain briefly how perfect hashing works. Separately, what is the situation when a new key cannot be inserted in a Cuckoo hash table separately?
3. Show your construction of an optimal Huffman code for the set of 10 frequencies: **a**:2 **b**:6 **c**:5 **d**:8 **e**:13 **f**:34 **g**:34 **h**:15 **i**:27 **j**:9
4. Given a weighted directed graph $G = (V, E, w)$ and a shortest path P from s to t, if the weight of every edge is doubled to produce $G^* = (V, E, w^*)$, is P still a shortest path in G^*? Explain your reasoning behind your answer.
5. BFS (breadth-first search) and DFS (depth-first search): Give the visited node order for each type of graph search, starting with S below.



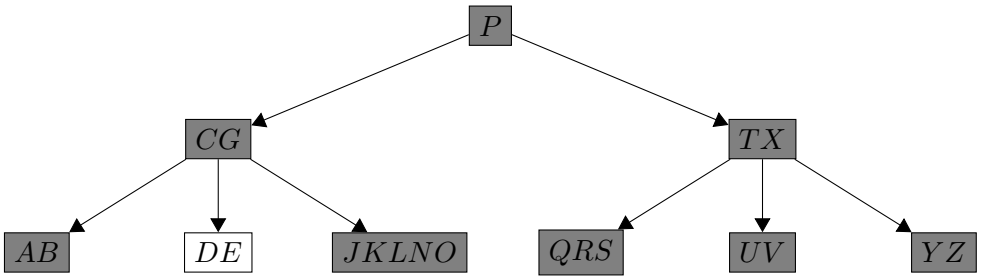
6. Many problem have been proved to be NP-complete. To prove NP-completeness, it is necessary in general to demonstrate two proof components. What are the two proof components to show a problem being NP-complete?

   Being NP-complete, the traveling-salesman problem (TSP) has a 2-approximation solution in polynomial time based on establishing a minimum spanning tree (MST) rooted at the start/end vertex (in polynomial time following MST-PRIM), if the graph edge weights observe the triangle inequality. Sketch a brief proof to demonstrate that that such a proof satisfies

2-approximation.

9.1.2 Long Questions (Answer all four questions.)

1. Given a B-tree with the minimum degree of $t = 3$ below, show the results after (i) deleting B, (ii) followed by inserting M, (iii) then followed by deleting T, and then inserting M_t for $M < M_t < N$



2. Knapsack problem: Suppose you want to pack a knapsack with weight limit W. Item i has an integer weight w_i and real value v_i. Your goal is to choose a subset of items with a maximum total value subject to the total weight $\leq W$.
Let $M[n, W]$ denote the maximum value that a set of items $\in \{1, 2, \ldots, n\}$ can have such that the weight is no more than W. We have the following recursive formula.

$$M[n,W] = \begin{cases} 0 & \text{if } n = 0 \text{ or } W = 0 \\ M[n-1,W] & \text{if } w_n > W \\ \max(M[n-1,W-w_n] + v_n, M[n-1,W]) & \text{otherwise} \end{cases}$$

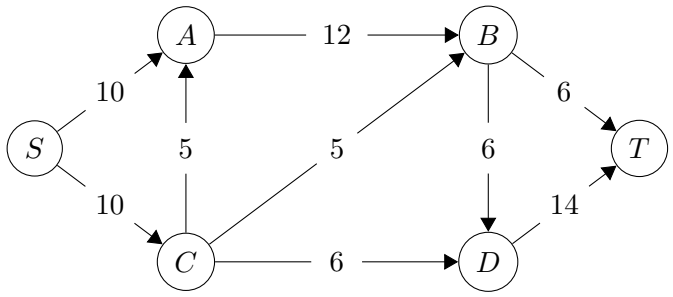
The time complexity of a simple recursive procedure as given below is exponential.

```
M(n,W)
{
        if (n==0 or W==0)
                return 0;
        if (w_n > W)
                result = M(n-1,W);
        else
                result = max{v_n + M(n-1,W-w_n), M(n-1,W)};
        return result;
}
```

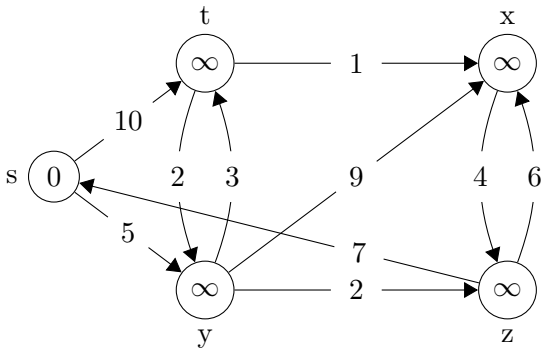
Provide a dynamic programming solution of the Knapsack problem after adding two lines of code to the above procedure. (Hint: Use a table to memorize the results.)

3. Follow the Ford-Fulkerson Algorithm to compute the max flow of the flow network illustrated below. Show each step to compute the max flow and also show the min cut of the flow

network.



4. The Dijkstra's algorithm (DIJ) solves the single-source shortest-path problem in a weighted directed graph $G = (V, E)$. Given the graph G below, follow DIJ to find shortest paths from vertex s to all other vertices, with all predecessor edges shaded and estimated distance values from s to all vertices provided at the end of each iteration.
   What is the time complexity of DIJ for a general graph $G = (V, E)$, if the candidate vertices are kept in a binary min-heap?



9.1.3 Solutions

Short problem #1

$$\begin{aligned}
\log(n!) &= \log(n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1) \\
&= \log(n) + \log(n-1) + \log(n-2) + \cdots + \log(2) + \log 1 \\
&\leq \log(n) + \log(n) + \log(n) + \cdots + \log(n) + \log(n) \\
&= n \log n \\
n \log(n!) &\leq n \cdot n \log(n) = n^2 \log n
\end{aligned}$$

Since the $n \log(n!) = O(n^2 \log n)$ is the term of largest asymptotic growth, $f(n) = O(n^2 \log n)$.

Short problem #4 To be the "shortest path P from s to t" means that it is the path of shortest length, and the length means the sum of the weights of the edges that P traverses. Consider any path from v_0 to v_k, $\langle v_0, v_1, v_2, \ldots, v_k \rangle$. The length of this path is $w(v_0)+w(v_1)+\cdots+w(v_k)$.

If all of the edge weights are doubled, the length of each path is doubled, so the path of shortest length under w will still be the shortest under w^*

Short problem #5 So many answers to this problem.

BFS: S, A, D, C, B or S, D, A, B, C

DFS: S, A, B, C, D or S, D, C, A, B

Short problem #6a The two proof components to show that a problem B is NP-complete are

1. An NP-complete decision problem A, and
2. A polynomial-time transformation that maps instance of A to instances of B.

The decision problem B is therefore at least as hard as A, because if B were easier, one could solve any instance of A by transforming it into an instance of B and using the solution method for B.

Short problem #6b Prove that there is a 2-approximation solution in polynomial time to the Traveling Salesman Problem if the weights satisfy the triangle inequality.

Let an actual solution to TSP be the tour $H*$.

Let the approximate solution be H.

To make H:

1. Grow a minimal spanning tree T with some root a, which either Kruskal or Prim can do in polynomial time. Since it is a minimal spanning tree and has no cycles, its cost cannot be greater than the tour of least cost, so $c(T) \leq c(H*)$.
2. From the tour, create a walk W starting at, and returning to, a. Since it traverses each edge twice, $c(W) = 2c(T)$.
3. Delete from the walk the second appearance of any vertex except a to create the Hamiltonian walk H of a preorder tree walk of T. Because the edge weights satisfy the Triangle Inequality, deleting a vertex does not increase the cost, so $c(H) \leq c(W)$.
4. Since $c(H) \leq c(W) = 2c(T) \leq 2c(H*)$, the Hamiltonian walk we have created in polynomial time has total cost no more than twice the cost of a perfect solution to the TSP, and thus we have a 2-approximation solution in polynomial time.

9.2 Fall 2018

9.2.1 Short Questions (Answer five of six.)

1. The divide and conquer strategy (D&C) has been used to solve problem efficiently to reduce the overall computational cost to certain types of problems.

   a. Which conditions have to be satisfied for D&C to solve such problems successfully? (Clearly state.)

   b. Suppose the size of a problem involved in D&C is $n = 2^k$. Let the cost in dividing the problems into an equal size is constant and the time to combine solutions to sub-problems is linear. Write the recurrence relations and then find the tight bound in solving such

problems using D&C.

2. a. What is the lower bound for comparisons based sorting algorithm? (Outline the justification of your answer.)
   b. What is the strategy behind greedy algorithm?
3. Find the tight bounds (by deriving their upper and lower bounds) of the following expressions.
   a. $T(n) = 2 \cdot T\left(\frac{n}{8}\right) + n^{\frac{1}{3}}$
   b. $T(n) = log(n!)$
4. Briefly describe what is dynamic programming. Can one apply it to solve any optimization problem? If yes, explain why; if not, what particular conditions must be met.
5. The utilization efficiency of a hash table depends heavily on its hashing function(s) employed. Describe with a diagram to illustrate how a multiplication method of hashing functions works on a machine with the word size of w bits for a hash table with 2^p entries, $p < w$.
6. Show your construction of an optimal Huffman code for the set of 7 frequencies: $\mathbf{a} : 3, \mathbf{b} : 12, \mathbf{c} : 5, \mathbf{d} : 20, \mathbf{e} : 16, \mathbf{f} : 34, \mathbf{g} : 18$

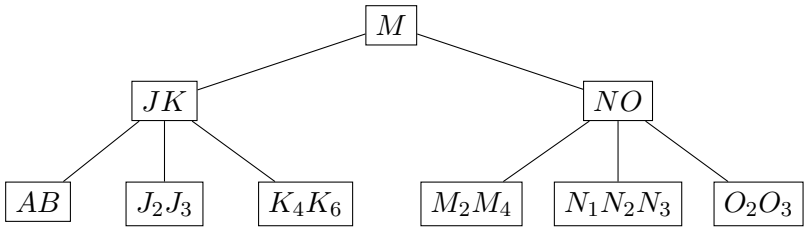
9.2.2 Long Questions (Answer four of five.)

1. Describe how dynamic programming is used to construct an optimal binary search tree.
   Use the following probability of p and q, obtain the expected cost of searching an optimal binary search tree constructed by dynamic programming.

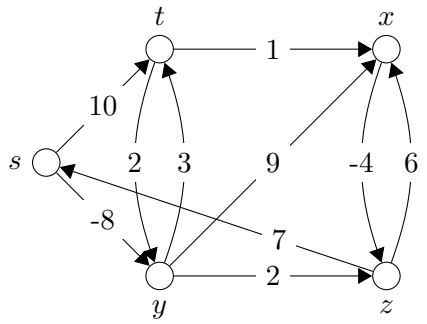
i	0	1	2	3	4	5	6	7
p_i		0.04	0.06	0.08	0.06	0.10	0.12	0.10
q_i	0.06	0.06	0.06	0.06	0.05	0.05	0.05	0.05

2. Given the initial B-tree with the minimum node degree of $t = 3$ below, show the results
   a. After deleting the key of K,
   b. Followed by inserting the key of L,
   c. Then by deleting the key of J_2,
   d. Then by inserting the key of N_4 with $N_3 < N_4 < O$, and
   e. Then by deleting K_4.

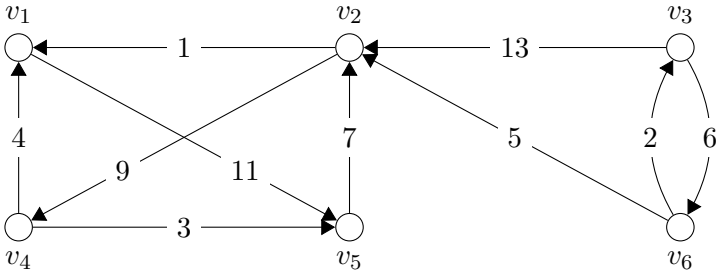
   (Show the result after every deletion and after every insertion.)



3. Follow depth-first search (DFS), starting from Node s, to traverse the graph shown below, with its edge weights all ignored and the start time equal to 1. Mark (1) the type of every edge and (2) the discovery and finish times of each node.
   Follow breadth-first search (BFS), starting from Node s, to traverse the graph shown below, with its edge weights all ignored. Show the predecessor tree rooted at Node s after BFS, with the number of links (i.e. distance) from Node s to every other node indicated.



4. The Dijkstra's algorithm (DS) solves the single-source shortest-path problem in a weighted graph $G = (V, E)$ without negative weighted edges or cycles, by edge relaxation at one vertex at a time until all vertices are examined. Given the graph G below, follow DS to find the shortest paths from vertex v_1 to all other vertices, with all predecessor edges shaded and estimated distance values from v_1 to all vertices provided at the end. Also list the sequence of vertices at which relaxation takes place.
   What is the time complexity of DS for a general graph $G = (V, E)$ when candidate vertices are kept in an array?



5. Given the matrix-chain multiplication problem for four matrices sized 30×24, 24×15, 15×20, 20×50, follow the tabular, bottom-up method in the procedure of `MATRIX-CHAIN-ORDER` below to construct two tables of $m[i, j]$ for all $1 \le i, j \le 4$, and $s[i, j]$ for all $1 \le i \le 3$ and $2 \le j \le 4$. Construct the two tables, with their entry values shown.

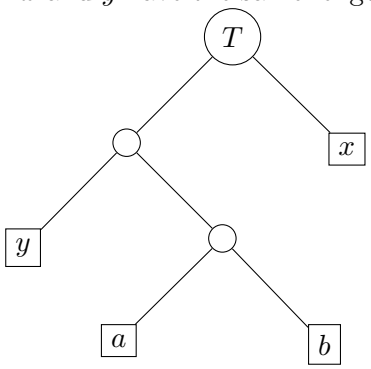
```
1  n = p.length − 1
2  let m[1..n, 1.n] and s[1..n − 1, 2..n] be new tables.
3  for i = 1 to n
4          m[i][j] = 0
5  for l = 2 to n // l is the chain length.
6          for i = 1 to n − l + 1
```

```
7                   j = i + l − 1
8                   m[i][j] = ∞
9                   for k = i to j − 1
10                          q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11                          if q < m[i, j]
12                                  m[i, j] = q
13                                  s[i, j] = k
```

9.3 Spring 2018

9.3.1 Short Questions (Answer six of seven.)

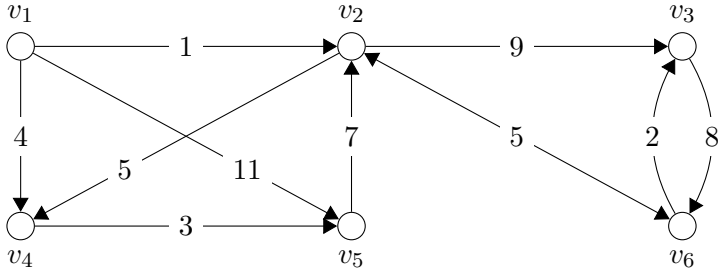
1. A problem with size n follows a typical divide-and-conquer approach to have its time complexity of
$$T(n) = T\left(\frac{n}{4}\right) + T\left(\frac{3n}{4}\right) + c \cdot n$$
Solve $T(n)$. (Show your work.)
2. Given that for an open-address hash table with load factor $\alpha = \frac{n}{m} < 1$, the expected number of probes in unsuccessful search under uniform hashing is at most $\frac{1}{1-\alpha}$, how do you prove the expected number of probes in a successful probe under uniform hashing being at most $\frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$? (Just give a proof sketch, explaining how many probes are needed to locate existing keys.)
3. (Lemma 16.2, §16.3, page 433) Sketch a proof of the Lemma below, using the tree provided. Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.



4. The Dijkstra's algorithm (DS) solves the single-source shortest-path problem in a weighted graph $G = (V, E)$ without negative weighted edges or cycles, by edge relaxation at one vertex at a time until all vertices are examined. Given the graph G below, follow DS to find the shortest paths from vertex v_1 to all other vertices, with all predecessor edges shaded and estimated distance values from v_1 to all vertices provided at the end. Also list the sequence

of vertices at which relaxation takes place.

What is the time complexity of DS for a general graph $G = (V, E)$ when candidate vertices are kept in an array?



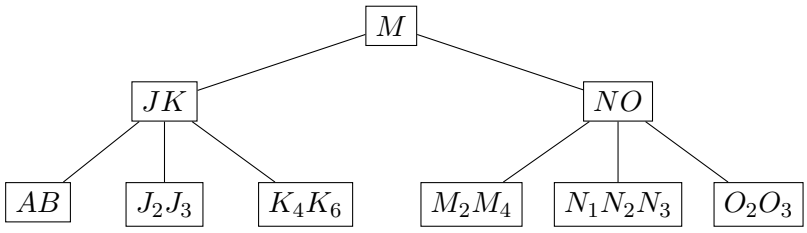
5. a. Define height balanced binary tree
   b. Write a pseudo code to determine whether a tree is height balanced?
   c. Obtain a tight bound of your algorithm.

9.3.2 Long Questions (Answer three of four.)

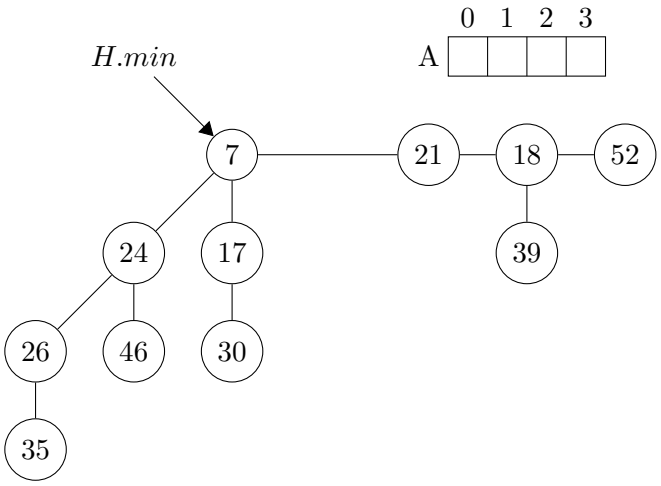
1. Given the initial B-tree with the minimum node degree of $t = 3$ below, show the results

   a. After deleting the key of M_2,
   b. Followed by inserting the key of L,
   c. Then by deleting the key of J_2,
   d. Then by inserting the key of O_1 with $O < O_1 < O_2$, and
   e. Then by deleting K.

   (Show the result after every deletion and after every insertion.)



2. A Fibonacci min-heap relies on the procedure of CONSOLIDATE to merge min-heaps in the root list upon the operation of extracting the minimum node. Given the following Fibonacci min-heap, show every consolidation step and the final heap result after $H.min$ is extracted, with the aid of A.



3. a. To what extent the asymptotic upper bound and lower bound provide insight on running time of an algorithm.
   b. Compare and contrast asymptotic tight bound to the average running time of an algorithm.
   c. Consider the pseudo code of an algorithm given below.
      i. What the value K in line 4 denote?
      ii. What the value m in line 8 denote?
      iii. When the algorithm terminates, what does the value $m + K$ in line 9 denote?
      iv. Find the asymptotic tight bound of Algorithm Test below.

```
AlgorithmTest(n)
1  K = 0
2  for i = 1 to n
3          for j = 1 to i
4                  K = K + 1
5  m = 0
6  for i = 1 to n − 1
7          for j = i + 1 to n
8                  m = m + 1
9  return (m + K)
```

4. a. Define the following classes of a decision problem: P, NP, and NP-completeness.
   b. Consider the 0-1 knapsack problem with n objects each with its respective pre-defined profit. The objective is to maximize the total profit that can be accommodated into a container of capacity W. Define appropriate notations for weight and profit of objects, formulate the problem.
   c. Convert of the problem that you have defined in (b) into a decision problem.
   d. Show the problem that you have defined in (c) belongs to NP-class.
   e. Does the problem in (d) belong to the P-class or NP-completeness. (Justify your answer.)
   f. If principle of optimality be applicable to solve the problem defined in (c), formulate it.

Otherwise, explain why not.

g. What would be your explanation, if 0-1 knapsack problem is solved by dynamic programming in polynomial time?

9.4 Spring 2017

9.4.1 Short Questions (Answer all six.)

1. a. Define height balanced binary tree.
   b. Write a pseudo code to determine whether a tree is height balanced?
   c. Obtain the tight bound of your algorithm.
2. Suppose you have keys of N objects stored in an array in the ascending order of key values. Also assume that there is duplicate entry in the key.
   a. Describe an efficient algorithm with the pseudo code that helps you to search for the object given the key. (The algorithm must return null value if the key is not in the array.)
   b. Obtain the tight upper bound for your algorithm.
3. a. Define strongly connected components as applied to directed graphs.
   b. Some potential application of strongly connected components.
   c. Provide a pseudo code for obtaining strongly components of a directed graph. (Hint: Use depth first search on appropriately transformed graphs.)
4. Find the tight for the recurrence relation below without using the master theorem (show all the steps):
$$T(n) = T(n/2) + n$$
5. a. What are the properties of min heap and max heaps.
   b. What is the preferred data structure of implementing binary heap, also justify your answer.
   c. What is the time complexity of merging two different min heaps each of size n and m.
6. Briefly describe the minimum spanning tree. Sketch an algorithm to obtain a minimum spanning tree.
   a.

9.4.2 Long Questions (Answer three of four.)

1. (Problem is related to dynamic programming formulation and optimal sub structure)
   a. Explain what do you understand by "principle of optimality."
   b. Consider the problem of finding the longest common subsequences (LCS) in a pair of sequences, namely $X(x_1, x_2, \ldots, x_m)$ and $Y(y_1, y_2, \ldots, y_n)$

i. A brute force approach is to generate all possible subsequences of X and see whether it is also a subsequence of Y. What is the number of possible subsequence of X?

ii. Obtain the optimal substructure of an LCS.

c. Consider a chain matrix multiplication problem, $m_1 \times m_2 \times \cdots \times m_n$. It associative, but not commutative.

i. A brute force approach is to generate all possible ways of parenthsizing the matrix chain and compute the total number of operations. What is the number of possible grouping of the matrix chain?

ii. Obtain the structure of the optimal parenthesization and then the recursive definition of the minimal cost of parenthesizing the product m_i, m_{i+1}, m_j.

2. (Similar to S15 #L3)

a. Compare and contrast P, NP, NP-complete, and NP-hard.

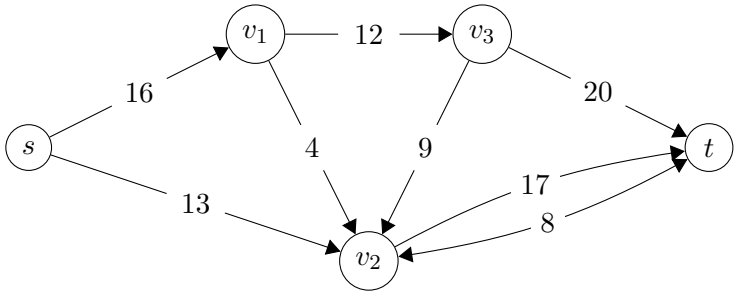
b. Based on current conjecture, draw a Venn diagram to show the relationship among these classes of problem.

c. Suppose there n clauses and m propositions in a given 3p-sat problem. How many possible interpretations are there? What is the time complexity of testing the satisfiability of a given interpretation? What is the time and space complexity of testing the satisfiability of the clauses?

d. 3-p sat problem is NP-complete, but people still have published papers by applying heuristics strategy and showing that they were able to solve it with large number of distinct propositions (say 100) and large number of clauses (say 200). To avoid any bias, they have generated the clauses and the set of propositions randomly. How would you start investigating their results? Can their results be generalized?

e. Suppose a single NP-complete problem is solved in polynomial algorithm, what can you state about the entire NP-complete class as well as the NP-hard class.

3. (Same instructions, different graph, as S15 #S7) The Edmonds-Karp Algorithm (EK) follows the basic Ford-Fulkerson method with breadth-first search to choose the shortest augmenting path (in terms of the number of edges involved) for computing the maximum flow iteratively from vertex s to vertex t in a weighted directed graph. Illustrate the maximum flow computation process (including the augmenting path chosen for each iteration and its resulting residual network) via EK for the graph depicted below.



SHOULD THE EDGE WITH WEIGHT 8 BE BIDIRECTIONAL?

4. (Same as Fall 2016 #L4 and S15 #L4)
   Given a set of 4 keys, with the following probabilities, determine the cost and the structure of an optimal BST (binary search tree), following the tabular, bottom-up method realized in the procedure of `OPTIMAL-BST` below to construct and fill $e[1..5, 0..4], w[1..5, 0..4]$ and $root[1..4, 1..4]$.

i	0	1	2	3	4
p_i		0.10	0.08	0.22	0.21
q_i	0.06	0.12	0.07	0.05	0.09

   Construct and fill the three tables, and show the optimal BST obtained.

```
OPTIMAL-BST(p,q,n)
1  let e[1..n+1,0..n], w[1..n+1,0..n], and root[1..n,1..n] be new tables.
2  for i = 1 to n+1
3          e[i,i-1] = q_{i-1}
4          w[i,i-1] = q_{i-1}
5  for l = 1 to n
6          for i = 1 to n-l+1
7                  j = i+l-1
8                  e[i,j] = ∞
9                  w[i,j] = w[i,j-1] + p_j + q_j
10                 for r = i to j
11                         t = e[i,r-1] + e[r+1,j] + w[i,j]
12                         if t < e[i,j]
13                                 e[i,j] = t
14                                 root[i,j] = r
15 return e and root
```

9.5 Fall 2016

9.5.1 Short Questions. (Answer eight of ten.)

1. (a) Define an upper and the tight time bound of an algorithm.
   (b) In which way the average time bound will add more value to the tight bound?
2. (a) Briefly describe a quick sort algorithm for sorting objects in an ascending order of their keys.
   (b) What is the best and worst case time complexity of quick sort and the reason for such complexity?
3. Find the tight [time bound, time complexity?] for the recurrence relations without using the master theorem.

(a) $T(n) = T(n-2) + 2\lg(n)$

(b) $T(n) = T(\sqrt{n}) + \lg(n)$

4. (a) What are the properties of min heap and max heaps.

   (b) What is the preferred data structure of implementing binary heap, also justify your answer.

   (c) What is the time complexity of merging two different min heap with sizes of n and m.

5. Suppose there are n clauses and m variables (propositions) in a given $3-p$ sat problem.

   - How many possible interpretations are there?
   - Find the tight bound of checking for satisfiability of the n clauses.

6. (a) Briefly define greedy strategy.

   (b) Will it always yield an optimal solution? If not, provide an algorithm that yields an optimal solution.

7. (a) Define strongly connected components.

   (b) Does a strongly connected component graph cyclic or acyclic? Justify your answer.

8. Mark true/false (T/F) against the following statements.

   - A binary search tree of size N will always find a key at most $O(\lg N)$ time
   - A breadth first search can be considered as a special case of heuristic search algorithm.
   - An optimal binary search tree is not necessarily being a balanced tree.
   - A dynamic programming approach uses top-down problem solving strategy to solve optimization problem.

9. Given two hash functions h_1 and h_2 for Cuckoo hashing under two tables, T_1 and T_2,

   - Describe the steps involved in inserting a record with the key of K_{new}.
   - Cuckoo hashing can be analyzed by the Cuckoo graph, whose nodes denote table entries and links connect pairs of nodes where given keys can be held. State when a new key can be inserted successfully based on the Cuckoo graph.

10. The recurrence of Procedure CUT-ROD(p, n) is given by

$$T(n) = 1 + \sum_{j=1}^{n-1} T(j),\ T(0) = 1$$

    Solve $T(n)$.

9.5.2 Notes

#5. According to Long Problem #1, "3-p sat" means "three-proposition satisfiability." There is a field of Boolean satisfiability problems called 3-sat. It's an NP-complete problem.

#7. Strongly connected components in Cormen §22.5

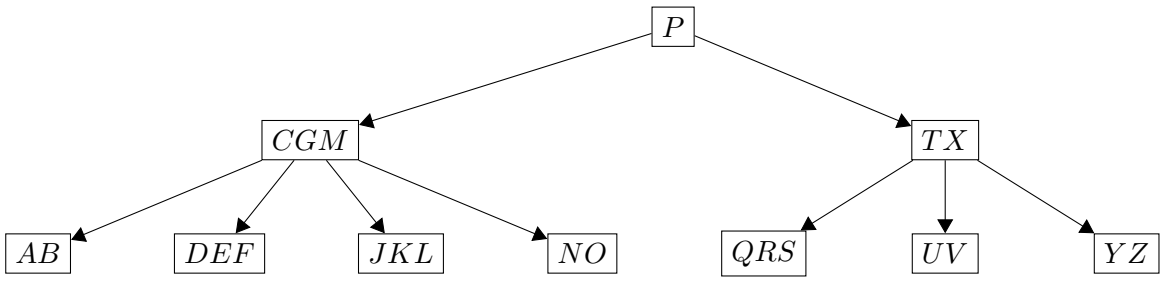
#8. "Heuristic" not in Cormen.

#9. "Cuckoo hashing" not in Cormen.

#10. $T(n) = 2^n$.

9.5.3 Long Questions. (Answer three of four.)

1. (a) Briefly describe NP-class, P-class, NP-complete, and NP-hard.
   (b) Show the conjectured relationship among the classes NP-class, P-class, NP-complete, and NP-hard.
   (c) Show that counting n objects with integer key values belongs to NP-class.
   (d) Provide the steps involved in showing whether a problem belongs to NP-complete or not.
   (e) Illustrate the steps in step d by showing 3 proposition satisfiability (3-p sat) problem belongs to NP-complete.
   (f) Provide a pseudo code that attempt to solve 3-p sat problem heuristically.
2. (a) Explain what do you understand by "principle of optimality" in the context of dynamic programming.
   (b) Characterize 0-1 knapsack problem in terms of objective function, constraints, and the time and space complexity. (Assume there are n objects. Suppose an object i has weight w_i and profit p_i. The overall capacity of the container is W).
   (c) Show the 0-1 knapsack problem belong to NP-class.
   (d) Does it belong to P-class? (Provide an explanation accordingly.)
   (e) Write down the basic rule that satisfy the principle of optimality and domain related constraints to the following problems:
      i. 0-1 knapsack problem
      ii. Pairwise shortest path problem
3. Given an initial B-tree with the minimum node degree of $t = 2$ below, show the results
   (a) after inserting the key of H, and
   (b) then followed by deleting two keys in order: X then P. (show the result after insertion and the result after each deletion.)



4. (Same as S15 #L4 and S17 #L4) Given a set of 4 keys, with the following probabilities, determine the cost and the structure of an optimal binary search tree, following the tabular, bottom-up method realized in the procedure of `OPTIMAL-BST` below to construct and fill tables $e[1..5, 0..4]$, $w[1..5, 0..4]$, and $root[1..4, 1..4]$.

i	0	1	2	3	4
p_i					
q_i					

```
let e[1..n+1, 0..n], w[1..n+1, 0..n], and root[1..n, 1..n] be new tables.
for i = 1 to n+1
        e[i, i-1] = q_{i-1}
        w[i, i-1] = q_{i-1}
for l = 1 to n
        for i = 1 to n-l+1
                j = i+l-1
                e[i, j] = ∞
                w[i, j] = w[, i, j-1] + p_j + q_j
                for r = i to j
                        t = e[i, r-1] + e[r+1, j] + w[i, j]
                        if i < e[i, j]
                                e[i, j] = t
                                root[i, j] = r
return e and root
```

9.5.4 Notes

#4 is the same as Cormen 15.5-2

1.

9.6 Fall 2015

9.6.1 Short Questions (Answer all eight.)

1. Show that for arbitrary real constants a and b with $b > 0$, we have $(n + a)^b = \Theta(n^b)$.
2. Use the recursion-tree technique to derive the tight lower and upper bounds of the recursion $T(n) = T(n/3) + T(2n/3) + cn$.
3. What is the time complexity of insertion sort algorithm? Suppose you already know the total number of keys and range of the key values. What would be the best and worst results you will get when sorting n items? In addition to the previous information, if you already know that the key values are uniformly distributed, what would be your best and worst results? (Make sure you sketch the algorithm and provide rationale of your expected results. Do NOT derive the result.)
4. Define minimum spanning tree. What are the salient features of a minimum spanning tree? Name a couple of [two] algorithms that will help to find the minimum spanning tree from a graph. Out of the two, which one would you prefer and specify why.

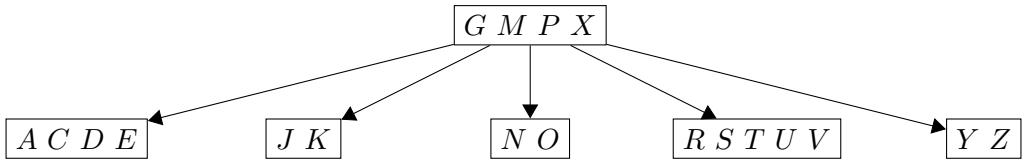
5. Mark True or False for each of the following statements.
   a. Greedy strategy sometimes finds the best or the optimal solution.
   b. Dynamic programming will always find the optimal solution even when the principal of optimality condition is not satisfied.
   c. (Same as S5 #S9b) Breadth first search is a special case of heuristic search algorithm.
   d. Some problems that belong to NP-class can be solved polynomially.
   e. Satisfiability problem of propositional calculus is NP-complete.
6. Obtain the optimal parenthesize for a chain multiplication given the S matrix as has been explained in the textbook or in class. (Show the working details.)

		[some letter] →					
		1	2	3	4	5	6
$d\uparrow$	7	4	4	4	3	5	6
	6	4	4	4	3	5	
	5	3	3	4	4		
	4	1	2	3			
	3	1	2				
	2	1					

7. Mark True or False against the following statements.
   a. (Same as S15 #9a) A binary search tree of size N will always find a key in at most $O(\log N)$ time.
   b. An optimal binary search is not necessary a balanced tree.
   c. A binary heap always maintains a balanced tree as practical as it can be.
   d. To implement a priority queue binomial heap is preferred over binary heap.
   e. A graph formed by strongly connected components, a strongly connected components graph (SCC) is always a minimum spanning tree.
8. For any n-key B-tree of height h and with minimum node degree of $t \geq 2$, prove that h is no larger than $\log_t \frac{n+1}{2}$. (Hint: Consider the number of keys stored in each tree level.)

9.6.2 Long Questions (Answer three of four.)

1. The utilization efficiency of a hash table depends on its hashing function(s) employed. Describe with a diagram to illustrate how a multiplication method of hashing works on a machine with word size of w bits for a hash table with 2^p entries, $p < w$. Explain briefly how Cuckoo hashing works under two hash functions of h_1 and h_2.
2. Given the initial B-tree with the minimum node degree of $t - 3$ below, show the results (a) after inserting two keys in order: Q then W, and (b) followed by deleting two keys in order: Y then T. (Show the aggregate result after insertion and another result after deletion.)



3. In the following pseudo code,
   a. Write the formula to determine the number of add operations at line 5 when the algorithm terminates.
   b. What is the space complexity?
   c. Find the following time bounds of this algorithm: Upper, lower, and tight. (Must show all the details of your work.)

```
Algorithm Count
1  Cnt = 0
2  for i = 1 to n do {
3          for j = 1 to i^2 do {
4                  for k = 1 to j do {
5                          Cnt = Cnt + 1
6                  }
7          }
8  }
```

Note that I have corrected line 3 from `for j=i^2 to i do{`

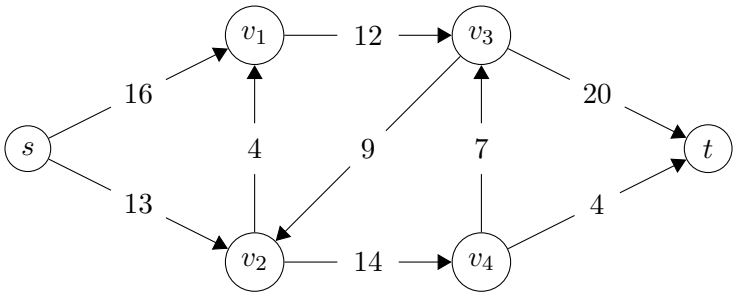
4. a. Describe maximum clique problem.
   b. Write a pseudo code to obtain the maximum clique of a graph with N nodes and E edges using generate and test strategy. That is, generate all possible subsets of vertices and test whether the subset is a clique. (Make any assumptions explicitly.)
   c. Find the time and space complexity of your algorithm.
   d. Describe the 0-1 knapsack problem.
   e. Show that the 0-1 knapsack problem belongs to NP-class.
   f. Briefly describe a practical way of solving a 0-1 knapsack problem and its time complexity assuming that the knapsack capacity is K and there are M objects. The weight and profit of an object i are denoted by w_i and p_i respectively.

9.7 Spring 2015

9.7.1 Short Questions

Answer all of the questions. All short questions, but Q3, carry 2 points each. Question 3 will be evaluated for 4 points.

1. Briefly define upper, lower, and tight time bound of an algorithm. How does an average time complexity related to any of these bounds?
2. In terms of run time efficiency, compare and contrast quick sort and merge sort. What is the best and the worst case time complexity of the quick sort algorithm? Also state under what conditions one may expect these two extreme cases.
3. Find the tight bounds of the following sums.
   (a) $\sum_{i=1}^{n} i^3 a^i$ where a is a constant greater than 1
   (b) $\sum_{i=1}^{n} \log(i^3)$
4. Mark true/false (T/F) against the following statements:
   (a) Connecting any pair of nodes in a minimum spanning tree will always form a cycle.
   (b) Building strongly connected component graph of a directed graph of N nodes takes $O(N^2)$ time.
   (c) A graph formed by strongly connected component nodes, a strongly connected component graph (SCC) is always a minimum spanning tree.
   (d) SCC graph will be useful in determining articulation node in a graph.
5. Use a binary tree representation to illustrate every operation of `MIN HEAPSORT` involved when sorting array $A = \{5, 13, 2, 25, 7\}$ without auxiliary storage.
6. Show your construction of an optimal Huffman code for the set of 7 frequencies: **a** : 2 **b** : 3 **c** : 5 **d** : 8 **e** : 13 **f** : 21 **g** : 34
7. (Same instructions, different graph, as S17 #L3) The Edmonds-Karp Algorithm (EK) follows the basic Ford-Fulkerson method with breadth-first search to choose the shortest augmenting path (in terms of the number of edges involved) for computing the maximum flow iteratively from vertex s to vertex t in a weighted directed graph. Illustrate the maximum flow computation process (including the augmenting path chosen for each iteration and its resulting residual network) via EK for the graph depicted below.



8. Briefly describe priority queue that maintains the highest key value. What is the preferred data structure for implementing priority queue? If the following keys with values 33, 22, 35, 40, 36, and 41 were added in the given order to an empty priority queue that maintains highest key value at the root, draw the priority queue at the end of all the given set of keys.

(Show all the key values and their indices.)

9. Mark true/false against the following statements.
   (a) (Same as F15 #S7a) A binary search tree of size N will always find a key in at most $O(\log N)$ time.
   (b) (Same as F15 #S5c) A breadth first search algorithm can be considered as a special case of heuristic search algorithm.
   (c) An optimal binary search tree is not necessarily a balanced tree.
   (d) A dynamic programming approach uses top-down problem solving strategy to solve optimization problem.

9.7.2 Long Questions (Answer three of four.)

1. (a) Suppose you have to find a solution to a problem that belongs to NP-complete class. Clearly summarize the steps that will help you to find the solution of the problem.
   (b) John, an undergraduate student, recently took data structure course, states that heuristics algorithms always solve NP-complete problems. He cites simplex methods as an example. If you agree with John, justify why or why not.
   (c) What is the strategy behind a greedy algorithm? Will it always provide an optimal solution? If yes explain why, otherwise say why not.
   (d) Consider the following pseudo code for Kruskal's algorithm for solving minimal spanning tree (MST).
   Algorithm MST. Let N be the number of nodes in graph G.

```
1 Sort the edges in non-decreasing order of cost.
2 T is an empty graph
3 while T has fewer than N-1 edges do:
4         let e denote the next edge of G (in the order of cost)
5         if T∪{e} does not contain a cycle, then T=T∪{e}
```

   Clearly mentioning the data structure you have to employ to reduce the time complexity to access and to maintain the necessary information, show the exact time taken to obtain the MST. Also show the tight bound of the algorithm. (Pay attention in detecting a cycle.)
2. (a) Explain what you understand by "principle of optimality."
   (b) Write down the basic rule that satisfies the principle of optimality and domain related constraints to the following problems.
      i. Knapsack Problem
      ii. Pairwise Shortest Path Problem
      iii. Chain Matrix Multiplication Problem
   (c) The optimal solution to the 0-1 knapsack problem belongs to NP-class. John says dynamic program formulation optimally solves the 0-1 knapsack problem. If you agree

with John, explain why; otherwise, explain why not.

3. (Similar to S17 #L2)
   (a) Compare and contrast P, NP, NP-Complete, and NP-Hard.
   (b) Based on current conjecture, draw a Venn diagram to show the relationship among these classes of problem.
   (c) Clearly state what is understood by propositional satisfiability problem.
   (d) Suppose there are m clauses and k propositions in a given 3-p sat problem. How many possible interpretations are there? What is the time complexity of testing the satisfiability of a given interpretation? What is the time and space complexity of testing the satisfiability of the clauses?
   (e) Suppose you came across a research paper that highlights a clever heuristic strategy that the authors have used to solve an instance of 3-p sat which has 10,000 propositions and 10,000 clauses with 3 propositions. They have generated several instances of the similar compositions of variables and clauses. The authors, in their point of view, have demonstrated the power of their heuristic to solve an instance of NP-complete problem in polynomial time (empirically shown). What would be your insight of their findings?

4. (Same as F16 #L4 and S17 #L4)
   (a) An object r is accessed by its key k_r. If all the objects have an equal chance of being accessed, what data structure will help you to have a better tight bound? (State your assumptions and provide the bound you have obtained for the proposed structure.)
   (b) An optimal binary search tree (`OPTIMAL-BST`) for a given set of keys with known access probabilities ensures the minimum expected search cost for key accesses, with its pseudo code listed below. Given the set of three keys with their access probabilities $k_1 = 0.25$, $k_2 = 0.15$, $k_3 = 0.3$, respectively, and four non-existing probabilities of $d_0 = 0.1$, $d_1 = 0.05$, $d_2 = 0.08$, $d_3 = 0.07$, construct optimal BST following dynamic programming with memorization for the given three keys and demonstrate the constructed optimal BST, which contains all three keys (k_1, k_2, k_3) and four non-exististing dummies (d_0, d_1, d_2, d_3). (Show your work using the three tables for expected costs, $e[i, j]$, for access weights, $w[i, j]$, and for $root[i, j]$, with i in $e[i, j]$ and $w[i, j]$ ranging from 1 to 4, j in $e[i, j]$ and $w[i, j]$ ranging from 0 to 3, and both i and j in $root[i, j]$ ranging from 1 to 3.)

```
OPTIMAL-BST(p,q,n)
1  let e[1..n+1,0..n], w[1..n+1,0..n], and root[1..n,1..n] be new tables.
2  for i = 1 to n+1
3          e[i,i-1] = q_{i-1}
4          w[i,i-1] = q_{i-1}
5  for l = 1 to n
6          for i = 1 to n-l+1
7                  j = i+l-1
8                  e[i,j] = ∞
```

```
 9                    w[i, j] = w[i, j − 1] + p_j + q_j
10                    for r = i to j
11                              t = e[i, r − 1] + e[r + 1, j] + w[i, j]
12                              if t < e[i, j]
13                                        e[i, j] = t
14                                        root[i, j] = r
15  return e and root
```