Solutions

1. A d-ary tree is a rooted tree in which each node has at most d children. Show that any d-ary tree with n nodes must have a depth of $\Omega(log\ n/log\ d)$. Can you give a precise formula for the minimum depth it could possibly have?

Solution: Any d-ary tree with n nodes must have a depth of $\Omega(log\ n/log\ d)$:
A d-ary tree of height h has at most $1 + d + \quad ... \quad + d^h = (d^{h+1} - 1)/(d - 1)$ vertices. We can prove this inductively.

(a) **Claim:** A d-ary tree of height h has at most $1 + d + \quad ... \quad + d^h = (d^{h+1} - 1)/(d - 1)$ vertices.

(b) **Base Case:** When the height of h of the tree is 1, the number of vertices is at most $1 + d = (d^{1+1} - 1)/(d - 1)$. Therefore, the statement is true when $h = 1$.

(c) **Induction Hypothesis:** The statement is true when the height h of the tree is $\leq k$.

(d) **Induction Step:** We will show that the statement is still true when $h = k + 1$. A d-ary tree of height $k + 1$ consists of a root that has at most d children, each of which is the root of a subtree with height at most k. By the Induction Hypothesis, the number of vertices in a d-ary tree of height $\leq k$ is at most $1 + d + \quad ... \quad + d^k = (d^{k+1} - 1)/(d - 1)$. Therefore, a tree with height $k + 1$ has at most $1 + d(1 + d + \quad ... \quad + d^k)$ vertices. Hence a tree with height $k + 1$ has at most $(d^{k+2} - 1)/(d - 1)$ vertices. We have completed the induction.

The height of a tree is equal to the maximum depth D of any node in the tree.

For a tree of height h, the maximum number of vertices $n \leq (d^{h+1} - 1)/(d - 1)$. Therefore,

$$d^{h+1} \geq n \cdot (d - 1) + 1 \tag{1}$$

$$(h + 1) \geq log_d(n \cdot (d - 1) + 1) \tag{2}$$

$$D = h \geq (log_d(n \cdot (d - 1) + 1) - 1) \tag{3}$$

By the definition of Big-Ω, the maximum depth $D \geq (log_d(n \cdot (d - 1) + 1) - 1) = \Omega(log_d\ (n)) = \Omega(log\ (n)/log\ (d))$ is true.

To obtain the specific formula:
Let d = maximum number of children, h = height of tree
Geometric series:

$$\Sigma nodes\ of\ complete\ tree = \frac{1 - d^{h+1}}{1 - d}$$

Using $\Sigma nodes\ of\ complete\ tree = n$:

$$n(1 - d) - 1 = -d^{h+1}$$

$$n(d - 1) + 1 = d^{h+1}$$

$$log_d n(d - 1) + 1 = d^{h+1}$$

$$\frac{log\ [n(d - 1) + 1]}{log\ d} = h$$

This represents the height of a graph of a complete tree. To get the height of an incomplete tree, we take the ceiling of this equation. Incomplete levels of the tree will be rounded up (eg., $h = 2.334 \rightarrow h = 3$)

$$\lceil \frac{log\ [n(d - 1) + 1]}{log\ d} \rceil = h$$

2. Sort each group of functions in increasing order of asymptotic (Big-O) growth. If two functions have the same asymptotic growth, indicate this.

   (a) $2^{2^{10}}n$, $1/n$, $log\ n$, n^2, 2^{10n}, $n(n-1)/2$, $n\sqrt{n}$, $n\ log\ n$

Solution: $1/n < log\ n < 2^{2^{10}}n < n\ log\ n < n\sqrt{n} < n^2 = n(n-1)/2 < 2^{10n}$

- $\underline{n\ log\ n < n\sqrt{n}}$

  Solution: Show that $n\ log\ n \in \mathcal{O}(n\sqrt{n})$

$$\lim_{n\to\infty} \frac{n\ log\ n}{n\sqrt{n}} = \lim_{n\to\infty} \frac{log\ n}{\sqrt{n}}$$

  (Using L'Hopital's rule)

$$= \lim_{n\to\infty} \frac{1/2\sqrt{n}}{1/n} = \lim_{n\to\infty} \frac{2}{\sqrt{n}} = 0 < \infty$$

  AND, show that $n\sqrt{n} \notin \mathcal{O}(n\ log\ n)$:

$$\lim_{n\to\infty} \frac{n\sqrt{n}}{n\ log\ n} = \lim_{n\to\infty} \frac{\sqrt{n}}{log\ n}$$

  (Using L'Hopital's rule)

$$= \lim_{n\to\infty} \frac{1/n}{1/2\sqrt{n}} = \lim_{n\to\infty} \frac{\sqrt{n}}{2} \not< \infty$$

- $\underline{n^2 = \frac{n(n-1)}{2}}$

  Solution: Show that $n^2 \in \Theta(\frac{n(n-1)}{2})$ ($n^2 \in \mathcal{O}(\frac{n(n-1)}{2})$ and $\frac{n(n-1)}{2} \in \mathcal{O}(n^2)$)

  To show $n^2 \in \mathcal{O}(\frac{n(n-1)}{2})$: take $c = 4$, $N = 2$

  To show $\frac{n(n-1)}{2} \in \mathcal{O}(n^2)$: take $c = 1$, $N = 1$

   (b) $n^{\sqrt{n}}$, 2^n, n^{10}, 3^n, $n^{log\ n}$, $\binom{n}{n/2}$, $log(n^n)$, $n!$

Solution: $log(n^n) < n^{10} < n^{log\ n} < n^{\sqrt{n}} < \binom{n}{n/2} < 2^n < 3^n < n!$

- $\underline{n^{\sqrt{n}} < 2^n}$

  Hint: $n^{\sqrt{n}}$ can be written as $2^{log\ n \cdot \sqrt{n}}$.

- $\underline{3^n < n!}$

  Solution: Show that $3^n \in \mathcal{O}(n!)$:

$$\lim_{n\to\infty} \frac{3^n}{n!} = \lim_{n\to\infty} \frac{3 * 3 * 3 \cdots (\text{n times})}{n * (n-1) \cdots (\text{n terms})}$$

$$= \lim_{n\to\infty} \frac{3}{n} \cdot \frac{3}{(n-1)} \cdot \frac{3}{(n-2)} \cdots \frac{3}{3} \cdot \frac{3}{2} \cdot \frac{3}{1}$$

$$\leq \lim_{n\to\infty} \frac{3}{n} \cdot \frac{3}{3} \cdot \frac{3}{3} \cdots \frac{3}{3} \cdot \frac{3}{2} \cdot \frac{3}{1}$$
$$= \lim_{n\to\infty} \frac{3}{n} \cdot \frac{9}{2} = 0 < \infty$$

AND, show that $n! \notin \mathcal{O}(3^n)$ by showing that

$$\lim_{n\to\infty} \frac{n!}{3^n} \not< \infty$$

- $\underline{\binom{n}{n/2} < 2^n}$

  Solution : Show that $\binom{n}{n/2} \in \mathcal{O}(2^n)$

  Using Stirling's approximation as $n \to \infty$,

$$\binom{n}{n/2} = \frac{n!}{(n/2)!(n/2)!} \approx \frac{\sqrt{2\pi n}\left(\frac{n}{e}\right)^n}{\left(\sqrt{\pi n}\left(\frac{n}{2e}\right)^{n/2}\right)^2} \approx \frac{\sqrt{2}\cdot 2^n}{\sqrt{\pi n}}$$

  Therefore,

$$\lim_{n\to\infty} \frac{\binom{n}{n/2}}{2^n} \approx \lim_{n\to\infty} \frac{\sqrt{2}\cdot 2^n}{\sqrt{\pi n}\cdot 2^n} = \lim_{n\to\infty} \frac{\sqrt{2}}{\sqrt{\pi n}} = 0$$

  Similarly, show that $2^n \notin \mathcal{O}\binom{n}{n/2}$

3. (a) Find (with proof) a function f_1 mapping positive integers to positive integers such that $f_1(2n)$ is $\mathcal{O}(f_1(n))$.

Solution: Consider $f_1(n) = n$.
Let $g(n) = f_1(n) = n$ and $h(n) = f_1(2n) = 2n$.
We now have to prove that $h(n) \in \mathcal{O}(g(n))$:
By the definition of Big-O, $h(n) \in \mathcal{O}(g(n))$ if $\exists$ positive constants c, N such that $h(n) \leq c \cdot g(n),\ \forall n > N$
Since, $2n \leq 2 \cdot n,\ \forall n > 0$ (here $c = 2$, $N = 0$), $h(n) \in \mathcal{O}(g(n))$

(b) Find (with proof) a function f_2 mapping positive integers to positive integers such that $f_2(2n)$ is not $\mathcal{O}(f_2(n))$.

Solution: Consider $f_2(n) = 2^n$.
Let $g(n) = f_2(n) = 2^n$ and $h(n) = f_2(2n) = 2^{2n} = (2^2)^n = 4^n$.
We now have to prove that $h(n) \notin \mathcal{O}(g(n))$:
Assume toward a contradiction that $h(n) \in \mathcal{O}(g(n))$. By the definition of Big-O, this means that there exist constants $c > 0$, N such that $h(n) \leq c \cdot g(n)\ \forall n \geq N$. But this is equivalent to saying $2^n \cdot 2^n \leq c \cdot 2^n\ \forall n \geq N \implies 2^n \leq c\ \forall n \geq N$. But the function 2^n is monotonically increasing in $n > 0$, and cannot be bounded by the assumed constant c for all $n > N$.

Alternate proof using limits:
By the definition of Big-O, $h(n) \in \mathcal{O}(g(n))$ if $\lim_{n\to\infty} \frac{h(n)}{g(n)} < \infty$

$$\lim_{n\to\infty} \frac{h(n)}{g(n)} = \lim_{n\to\infty} \frac{4^n}{2^n} = \lim_{n\to\infty} 2^n \not< \infty$$

(c) Prove that if $f(n)$ is $\mathcal{O}(g(n))$, and $g(n)$ is $\mathcal{O}(h(n))$, then $f(n)$ is $\mathcal{O}(h(n))$.

Solution: By the definition of Big-O, there exist positive integers N_1 and N_2 and positive constants c_1 and c_2 such that $f(n) \leq c_1 \cdot g(n)$ for $n > N_1$ and $g(n) \leq c_2 \cdot h(n)$ for $n > N_2$. Let $N_0 = max(N_1, N_2)$ and $c_0 = c_1 c_2$.

$$f(n) \leq c_1 \cdot g(n),\ \forall n > N_0 \quad (1)$$
$$g(n) \leq c_2 \cdot h(n),\ \forall n > N_0 \quad (2)$$

(1) and (2) imply,

$$f(n) \leq c_1 \cdot g(n) \leq c_1 c_2 \cdot h(n),\ \forall n > N_0$$

Thus, $f(n)$ is $\mathcal{O}(h(n))$

(d) Prove or disprove: if f is not $\mathcal{O}(g)$, then g is $\mathcal{O}(f)$.

Solution:
A counterexample can disprove the above claim:

Let $f(n) = 2^{n \cdot sin\ n}$, $g(n) = n$. We will show that $f(n) \notin \mathcal{O}(g(n))$ and $g(n) \notin \mathcal{O}(f(n))$

Definition: $f(n) \notin \mathcal{O}(g(n))$ if for all choices of positive c and N, there exists some $n > N$ such that $f(n) > cg(n)$.

Showing that $2^{n \cdot sin\ n} \notin \mathcal{O}(n)$:
Choose any c, $N > 0$. We can choose $n > N$ such that:

i. $sin\ n = d > 0$

ii. $2^{dn} > cn$

Similarly, show that $n \notin \mathcal{O}(2^{n \cdot sin\ n})$:
Choose any c, $N > 0$. We can choose $n > N$ such that:

i. $sin\ n = e < 0$

ii. $cn > 2^{en}$

4. A binary tree is a rooted tree in which each node has at most two children. Show that in any binary tree the number of nodes with two children is exactly one less than the number of leaves. (Hint: induction.)

Solution: We will prove this by induction on number of nodes in a binary tree T.
Let $n_0(T)$ = number of leaves of binary tree T, $n_2(T)$ = number of nodes in T with 2 children

(a) **Claim:** A binary tree T with n vertices satisfies $n_0(T) - 1 = n_2(T)$

(b) **Base Case:** A binary tree with 1 vertex has 1 leaf and no vertices with 2 children. Hence, the condition is satisfied.

(c) **Induction Hypothesis:** A binary tree T' with n vertices satisfies $n_0(T') - 1 = n_2(T')$.

(d) **Induction Step:** We will show the condition to be true for a binary tree T with $n + 1$ vertices.

Let T be an arbitrary binary tree with $n + 1$ vertices. Let v be a leaf of the tree. Since the tree has more than one vertex, v is not the root and it therefore has a parent u. Let T' be obtained by deleting the leaf v.

Case 1 : If the parent u had 2 children.
Number of leaves in the tree reduces by 1, $n_0(T') = n_0(T) - 1$
Number of vertices with 2 children reduces by 1, $n_2(T') = n_2(T) - 1$
By induction hypothesis, we know $n_0(T') = n_2(T') + 1$
$n_0(T') = n_2(T') + 1 => n_0(T) = n_2(T) + 1$
Therefore, the condition holds true for T with $n + 1$ vertices.

Case 2 : If the parent u had 1 child. Now u becomes a leaf. Number of leaves and vertices with 2 children remain unchanged:
$n_0(T') = n_0(T)$
$n_2(T') = n_2(T)$
$n_0(T') = n_2(T') + 1 => n_0(T) = n_2(T) + 1$
Therefore, the condition holds true for T with $n + 1$ vertices.

Hence proved by induction.

5. In this problem the input includes an array A such that $A[0 \ldots n-1]$ contains n integers that are sorted into non-decreasing order: $A[i] \leq A[i+1]$ for $i = 0, 1, \ldots, n-2$. The array A may contain repeated elements, e.g. $A = [0, 1, 1, 2, 3, 3, 3, 3, 4, 5, 5, 6, 6, 6]$

(a) Describe carefully an algorithm $COUNT(A, x)$ that, given an array A, and an integer x, returns the number of occurrences of x in the array A. Your algorithm should be similar to binary search, and must run in $\mathcal{O}(log\ n)$.

Solution: First we will implement the function FIRST_INDEX$(A, x, low, high)$ which returns the smallest i such that $low \leq i < high$ and $A[i] \geq x$. This is done by tweaking the binary search algorithm so that it continues searching for the first index of the "key" even if it has already found one occurrence. If no such index exists, then the function returns $high$. We implement FIRST_INDEX as follows:

```
FIRST_INDEX(A, x, low, high)
if low = high
 return low
mid = (low + high)/2
if A[mid] < x
 then return FIRST_INDEX(A, x, mid + 1, high)
if A[mid] ≥ x
```

```
then return FIRST_INDEX(A, x, low, mid)
```

Then we implement COUNT(A, x) which will give the number of occurrences of x in array A. It does this as shown below:

```
COUNT(A,x)
return FIRST_INDEX(A, x+1, 0, n) - FIRST_INDEX(A, x, 0, n).
```

Running time:
There are two calls to FIRST_INDEX by COUNT. Therefore, running time of our algorithm is the running time of FIRST_INDEX.
Let $n = high - low$. If $n > 1$, then FIRST_INDEX makes a recursive call to either the left half or the right half of the array i.e. single subproblem of size $n/2$. This gives the recursive relation

$$T(n) = T(n/2) + \Theta(1)$$

which can be solved using the Master Theorem to give $T(n) = \Theta(log\ n)$.

(b) Prove that your algorithm always terminates.

Solution: We need only show that FIRST_INDEX terminates.
The algorithm terminates because every recursive call to FIRST_INDEX processes a smaller range of A (the difference $high - low$ strictly decreases), so every call will eventually terminate at the base case where $low = high$.
Proof by induction:

(a) **Claim:** The algorithm FIRST_INDEX terminates with any input of size $n,\ n \geq 0$

(b) **Base Case:** The algorithm terminates with input of size 0: since $low = high$ it terminates

(c) **Induction Hypothesis:** The algorithm FIRST_INDEX terminates with any input of size $n,\ \forall n \leq k,\ k \geq 0$

(d) **Induction Step:** We need to show that FIRST_INDEX terminates with input of size $k + 1$
The algorithm makes a recursive call on either the left half of the array or the right half of the array. The input sizes on both these recursive calls is $< k + 1$. By induction hypothesis we know that the algorithm terminates for input sizes $\leq k$. Hence the algorithm will terminate.

This completes the proof

(c) Prove that when your algorithm terminates, it terminates with the correct answer.

Solution: The function FIRST_INDEX terminates with the correct answer because if $A[mid]$ is less than x, then no index less than mid can contain any element that is at least x, so we can recurse to the right half. On the other hand, if $A[mid]$ is at least x, then the smallest index i such that $A[i]$ is at least x must be at most equal to mid, so we can recurse to the left half.

Finally, the difference between FIRST_INDEX($A, x+1, 0, n$) - FIRST_INDEX($A, x, 0, n$) must be the number of indices that contain exactly x.

6. You are given a vertex-weighted graph. Consider the following definitions.

   Independent set: a set of vertices in a graph, no two of which are adjacent.

   Weight of independent set: sum of weights of vertices in the set.

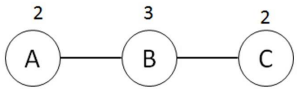
   Max-weight independent set problem: Find an independent set which has maximum weight.

   Consider the following "greedy" approach to finding a max-weight independent set in a given vertex-weighted graph:

   1) Start with an empty set X.
   2) For each vertex v of the graph, in decreasing order of weight:
      add vertex v to the set X if v is not adjacent to any vertex in X.
   3) Return X, $weight(X)$.

   (a) Show that the "greedy" approach for the max-weight independent set is not optimal, by exhibiting a small counterexample.

   Solution: Consider the graph shown below with $w(B) = 3$, $w(A) = w(C) = 2$:

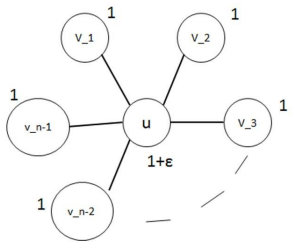
   

   The greedy approach adds B to set X and is unable to add more vertices. It returns $\{B\}$ with weight 3. However the max-weight independent set for the above is $\{A, C\}$ with weight $= 4$.

   (b) How badly suboptimal can greed be, relative to optimal? Please clearly explain your definition of "badly suboptimal".

   Solution: We define the suboptimality of greed relative to optimal as the ratio of the weight of the optimum max-weight independent set to the weight of the greedy max-weight independent set. The larger the ratio, the greater the suboptimality of the greedy approach.

   Consider the n-vertex "star" below:

   Worst case behavior of greedy approach:

   - the addition of first vertex u to X forbids other vertices from getting added (all other vertices are adjacent to u)

- Weight of u is just greater than that of the other vertices by a small amount (ϵ)

Therefore, suboptimality $= \frac{n-1}{1+\epsilon} \in \Omega(n)$.