Homework 3

Solution Key

Problem 1

Given a location i in the 1-indexed array A of length n, we can clearly tell if i is to the left of the max, to the right of the max, or *is* the max, via the following three-way test:

$$\begin{cases} i < n \textbf{ and } A[i] < A[i+1] & \rightarrow \quad \textit{max} \text{ IS TO THE RIGHT OF } i \\ i > 1 \textbf{ and } A[i] < A[i-1] & \rightarrow \quad \textit{max} \text{ IS TO THE LEFT OF } i \\ \textbf{else} & \rightarrow \quad \textit{max} \text{ IS AT } i \end{cases}$$

Knowing whether the desired location is to the left, to the right, or on the current location is exactly the subroutine used in the *binary search* algorithm, which will thus solve the problem in $O(\log n)$ time (since each test can be done in constant time).

[Note: this solution is significantly shorter than what was typically turned in. Study it carefully to see that we have included all the elements needed to clearly and convincingly demonstrate a) what we we are doing, b) why it works, and c) how long it takes.]

Problem 2

1. In all parts below we consider $k \leq n$, since no paths of length greater than n exist in the tree of size n.

   The structure of our algorithm is a recursive traversal of the tree, starting at the root, where we maintain, globally, the cost of the best path seen so far in a variable b. Consider a depth-first traversal of the tree, where for each node v we compute the cost $c(v)$ of the path to the root as the sum of the cost of its parent, and the cost of the edge to its parent; further, we keep track of the depth of recursion $d(v)$. For each node v traversed like this, if $d(v) = k$, then update $b \leftarrow \max\{b, c(v)\}$.

   Since this algorithm traverses the tree once, and involves constant work per node, the total time is $O(n)$, where n is the number of nodes in the tree.

2. The algorithm consists of two stages. In the first stage we produce two tables, L and R. For each integer $i \leq k$, the table $L[i]$ maintains the cost of the best path of length i to the root within the root's left subtree; $R[i]$ is defined analogously for the right subtree. This stage is an adaptation of the algorithm from part 1: as in part 1, traverse the tree such that at any node v we know its depth $d(v)$ and the cost of the path to the root $c(v)$. If v is in the left subtree, update $L[d(v)] \leftarrow \max\{L[d(v)], c(v)\}$, and if v is in the right subtree, update $R[d(v)] \leftarrow \max\{R[d(v)], c(v)\}$. Since all the vertices are traversed, the tables L and R will thus store the maximum costs of paths of each length, as claimed.

   Since the maximum path of length k through the root must consist of the union of a path of length i from the root down the left subtree, and a path of length $k - i$ from the root down the right subtree, for some $i \in \{0, \ldots, k\}$, in the second stage, we return the maximum, over $i \in \{0, \ldots, k\}$ of $L[i] + R[k - i]$.

The traversal of the tree takes time $O(n)$ as in part 1, and processing the arrays of length k takes time $O(k)$, for a total time of $O(n)$ since $k \leq n$.

3. Consider the subtrees of the root as being labeled by positive integers. This algorithm is similar to the algorithm for the previous part, however we now maintain three arrays: $B[i]$ stores the cost of the *best* path of length i to the root through any subtree, and $L[i]$ will store the label of the subtree through which this path goes; $D[i]$ will store the best path of length i that goes through a *different* subtree than $L[i]$.

   We construct these arrays by traversing the tree as in parts 1 and 2 so that at each node v we know its depth $d(v)$ and the cost of the path to the root $c(v)$; additionally, let $\ell(v)$ denote the label of the subtree where v lies. There are three cases:

   (a) **if** $\ell(v) = L[d(v)]$ **then** update $B[d(v)] \leftarrow \max\{B[d(v)], c(v)\}$;
   (b) **otherwise, if** $c(v) \geq B[d(v)]$ **then** update $D[d(v)] \leftarrow B[d(v)]$ and $B[d(v)] \leftarrow c(v)$ and $L[d(v)] \leftarrow \ell(v)$;
   (c) **otherwise, if** $c(v) \geq D[d(v)]$ **then** update $D[d(v)] \leftarrow c(v)$.

   For each case it is clear that we are correctly updating our three arrays, and that this is completed in constant time per node.

   Given arrays B, L, D the conclusion of the algorithm is to return the max, over $i \in \{0, \ldots, k\}$ of the following:

$$\begin{cases} B[i] + B[k-i] & if L[i] \neq L[k-i] \\ \max\{B[i] + D[k-i], D[i] + B[k-i]\} & \text{otherwise} \end{cases}$$

   Since each path through the root consists, for some $i \in \{0, \ldots, k\}$, of a path of length i descending from the root down one subtree, plus a path of length $k - i$ descending from the root down a *different* subtree, our algorithm will clearly return the maximum cost.

   The algorithm runs in time $O(n)$ since we traverse each node once, doing constant work per node, and the second step takes $O(k)$ time, where by assumption $k \leq n$.

4. Our algorithm has two steps. In the first step, for each vertex v we compute and store $s[v]$, the *size of the subtree* rooted at v, namely the number of nodes that are descendents of v plus v itself. We compute these sizes in linear time by first initializing $s[v] = 1$ for all vertices, and then conducting a depth-first traversal of the tree where, to process a node v with parent $p(v)$, we increment $s[p(v)] \leftarrow s[p(v)] + s[v]$. Provided the algorithm correctly computes $s[c_i]$ for each child c_i of a node v, then this update rule will correctly set $s[v] = 1 + \sum_i s[c_i]$; thus our algorithm is correct by induction up the tree.

   Given the array s, we then traverse down the tree starting from the root by descending down a subtree of size $> \frac{n}{2}$ if one exists, and returning the current node otherwise.

   When the algorithm terminates at node v, by construction each subtree below v will have size at most $\frac{n}{2}$, and we need only prove that the portion of the tree that is the complement of the subtree rooted at v also has size at most $\frac{n}{2}$. Consider the parent p of v. (If v has no parent then v is the root and we are done.) Since the algorithm did *not* terminate at p and chose to descend to v, the subtree rooted at v must have size $> \frac{n}{2}$, meaning that the complement has size $< n - \frac{n}{2} = \frac{n}{2}$, as desired.

As noted above, the running time for the first part of the algorithm is $O(n)$; the second part examines each entry of s at most once, doing constant work in each case, for a total time of $O(n)$.

5. (See part 4)

6. Consider a node v as returned by the algorithm from part 5, and consider an optimal path of length k in the tree. Further, consider making v the root of the tree (this can be done in linear time via a depth-first search starting at v). There are two cases: either the path passes through v, or the path does not pass through v in which case it must be contained entirely within a single subtree of v.

   This observation gives us the structure of the algorithm:

```
MAX-WEIGHT-PATH($G, k$)
1  $v$ = the vertex from the algorithm of part 5.
2  Rearrange $G$ to make $v$ the root.
3  return max $\begin{cases} \text{Result of the algorithm of part 3 on } (G,k) \\ \text{MAX-WEIGHT-PATH}(S_i,k) \text{ for each subtree } S_i \text{ of } v \end{cases}$
```

   As discussed above, given the correctness of algorithms from parts 3 and 5, the correctness of this algorithm follows.

   We now analyze the run time. Note that for each recursive invocation of the algorithm on a tree of size t, we do $O(t)$ work at the current level of recursion, namely a constant amount of work per vertex. Further, since the recursive calls are on *disjoint* sets of vertices, the total time spent at a given depth j of the recursion, across all recursive calls of MAX-WEIGHT-PATH, is $O(n)$. As shown in part 4, each recursive call will be on a subtree at most half the size, which implies that there are at most $\log n$ levels of recursion. Thus the total time of our algorithm is $O(n \log n)$, as desired.

Problem 3

1.
```
function out=myconv(a,b)
% This function takes as input row vectors "a" and "b", and returns their
% convolution in the variable "out". For example, myconv([1 2],[3 4 5])
% should return [3 10 13 10]

outputLength = length(a) + length(b) - 1;
aAddedLength = outputLength - length(a);
bAddedLength = outputLength - length(b);
a = [a zeros(1, aAddedLength)];
b = [b zeros(1, bAddedLength)];

out =  ifft(fft(a) .* fft(b));
```

2. Matlab stores numbers in *double precision*, by default, which stores roughly 16 significant digits for each number. Thus, if in Matlab you evaluate $10^{20} + 1 - 10^{20}$, Matlab will return 0, instead of 1, because $10^{20} + 1$ is the *same* as 10^{20} to 16 significant figures. Because of the

way Fourier transforms "mix" their inputs, if 10^{20} and 1 appear together in the same Fourier transform, the resulting Fourier transform may look as though we started with the pair 10^{20} and **0** instead. One example of this is: `myconv([1e20 1],1)`, whose second component will be **0** and not 1, which is pretty embarrassing.

3.
   ```
function out=roundup235(n)
% This function rounds up a number n to the next number that is a product
% of powers of 2, 3, and 5. For example, roundup235(999997) should return
% 1000000.

out=inf;
for i=0:log(n)/log(2)+1
    for j=0:log(n)/log(3)+1
        for k=0:log(n)/log(5)+1
            if 2^i*3^j*5^k>=n, out=min(out,2^i*3^j*5^k); end
        end
    end
end
```

4.
   ```
function out=myconv235(a,b)
% This function takes as input row vectors "a" and "b", and returns their
% convolution in the variable "out". For example, myconv([1 2],[3 4 5])
% should return [3 10 13 10]. Each of the Fourier transforms in this
% function has a size which is a product of powers of 2,3, and 5, as
% computed by roundup235

originalOutputLength = length(a) + length(b) - 1;
outputLength = roundup235(originalOutputLength);
aAddedLength = outputLength - length(a);
bAddedLength = outputLength - length(b);
a = [a zeros(1, aAddedLength)];
b = [b zeros(1, bAddedLength)];

out =  ifft(fft(a) .* fft(b));
out = out(1:originalOutputLength);
```

5. To find the best speedup, we look for two vectors whose convolution has a size that is a *large prime*. Matlab's `factor` command helps us find that, for example, 999983 is prime, which would be the size of a convolution of vectors of lengths 500000 and 499984. On Bilbo's machine, `tic; myconv235(ones(1,500000),ones(1,499984)); toc` reports 0.1 seconds, while `tic; myconv(ones(1,500000),ones(1,499984)); toc` takes 1.6 seconds, a 16× speedup! (Matlab's built-in function, `tic; conv(ones(1,500000),ones(1,499984)); toc` takes more than 30 seconds.)

6.
   ```
function out=bigmult(a,b)
% This function takes as input row vectors "a" and "b", and, treating each
% entry as a digit of a large number, returns the product of these two
% numbers in the same format. For example, bigmult([5 0],[1 4]) returns
% [7 0 0].
```

```
out = round(myconv235(a,b));
carry=0;
for i = length(out):-1:1
    out(i) = out(i) + carry;
    carry = floor(out(i) / 10);
    out(i) = mod(out(i),10);
end
if carry, out = [carry out]; end
```

7. What we are worried about is rounding errors from non-integer intermediate computations. We can see that the maximum error in the convolution seems to grow roughly linearly with the number of digits involved: for 10,000,000 digit numbers, we can see a typical error by evaluating:

   ```
   z=myconv235(9*ones(1,10000000),9*ones(1,10000000)); max(abs(z-round(z)))
   ```

   This returns roughly `5e-7`, indicating that if the numbers were a million times longer we would run into rounding errors. Thus this code would be ineffective with numbers of more than 10^{13} digits.