Problem Set 2 Solutions

(Exercises were not to be turned in, but we're providing the solutions for your own interest.)

Exercise 2-1. No, this does yield the intended result. In particular, $A[1]$ will never contain the original first element of A. However, many non-identity permutations ($(n-1)! - 1$ of them, in fact) preserve the first element.

Exercise 2-2. Any heap with 2^i through $2^{i+1} - 1$ elements has height i (this can be verified by induction), where the height of a heap containing just the root is 0. Since $i = \lfloor \lg n \rfloor$ for any i in the range $2^i, \ldots 2^{i+1} - 1$, the claim follows.

Exercise 2-3. It's actually quite difficult to analyze the number of comparisons exactly, because the BUILD-HEAP procedure does some complicated re-arrangements (in the case of an already-sorted array). However, each call to MAX-HEAPIFY takes the worst-case time of $\Omega(\log n)$ in both cases, so HEAPSORT takes $\Omega(n \log n)$ time.

Exercise 2-4. When all the elements have the same value, each partition is worst-case (the pivot element stays at one end of the array), so the running time is $\Theta(n^2)$.

Exercise 2-5. Correctness of the base cases ($i = j$ or $i + 1 = j$) is obvious. After the second recursive call, the largest third of the elements are in their proper positions (this can be seen by considering three cases for the starting position of an element that is among the largest third). Then the third recursive call sorts the smallest two-thirds of the elements, leaving the entire array sorted. A recurrence is $T(n) = 3T(2n/3) + \Theta(1)$. Solving by Master Theorem, we get a worst-case runtime of $T(n) = \Theta(n^{\log_{3/2} 3})$, and $\log_{3/2} 3 \approx 2.71$. In fact, the asymptotic running time is essentially independent of the input, i.e. all inputs are worst-case. This running time is much worse than the running time of any other sorting algorithm we have seen.

Problem 2-1. Large-integer multiplication.

(a) We assume that an n-bit integer is represented as an n-element array of bits, where the bits are in increasing order of significance (i.e., the least significant bit is the first element, etc.). The precondition of this procedure is that the two arrays U and V have the same length (this can be achieved by padding the shorter array with zero bits). We assume that there is a naive procedure ADD which adds integers in time linear in the total number of digits of all its arguments. The operator $||$ appends two arrays (in time linear in the total number of elements), and 0^i represents an i-element array of zeros, which can be initialized in time i.

```
MULTIPLY(U, V)
    $n \leftarrow \textit{length}[U]$
    $m \leftarrow \lfloor n/2 \rfloor$
    $\triangleright$ $P$ will store the product; initialize it to length $2n$, with all zero entries
    $P \leftarrow 0^{2n}$
    if $n = 1$
        $\triangleright$ base case
        $P[1] = U[1] \cdot V[1]$
        return $P$
    $\triangleright$ recursive case; build $A, B, C, D$
    $A \leftarrow U[m+1, \ldots, n]$
    $B \leftarrow U[1, \ldots, m]$
    $C \leftarrow V[m+1, \ldots, n]$
    $D \leftarrow V[1, \ldots, m]$
    $\triangleright$ multiply recursively, and append zeros to multiply by powers of 2
    $P_1 \leftarrow 0^{2m} \,||\, \text{MULTIPLY}(A, C)$
    $P_2 \leftarrow \text{MULTIPLY}(B, D)$
    $P_3 \leftarrow 0^{m} \,||\, \text{MULTIPLY}(A, D)$
    $P_4 \leftarrow 0^{m} \,||\, \text{MULTIPLY}(B, C)$
    $\triangleright$ combine the answers
    $P \leftarrow \text{ADD}(P_1, P_2, P_3, P_4)$
    return $P$
```

On inputs of length n, the procedure calls itself four times on inputs of length (roughly) $n/2$. All the array initialization, copying, and appending requires $\Theta(n)$ time, as does the call to ADD. Therefore the recurrence for this procedure's running time is (after invoking the sloppiness theorem):

$$T(n) = 4T(n/2) + \Theta(n).$$

Solving by the Master Theorem, we get $T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$.

(b) The only change we make is in the way we compute $ad + bc$. First we compute ac and bd, then we compute $ad + bc = (a + b)(c + d) - ac - bd$ by multiplying $a + b$ with $c + d$ and subtracting off the two products we already computed. The recurrence for this procedure is now $T(n) = 3T(n/2) + \Theta(n)$, which we can solve by the Master Method as $T(n) = \Theta(n^{\log_2 3})$, and $\log_2 3 \approx 1.585$.

Here is the pseudocode. We assume the existence of a procedure SUBTRACT(X, Y) which computes $X - Y$ in time linear in the total length of both arrays X, Y.

```
MULTIPLY(U, V)
    $n \leftarrow \textit{length}[U]$
    $m \leftarrow \lfloor n/2 \rfloor$
    ▷ $P$ will store the product; initialize it to length $2n$, with all zero entries
    $P \leftarrow 0^{2n}$
    if $n = 1$
        ▷ base case
        $P[1] = U[1] \cdot V[1]$
        return $P$
    ▷ recursive case; build $A, B, C, D$
    $A \leftarrow U[m+1, \ldots, n]$
    $B \leftarrow U[1, \ldots, m]$
    $C \leftarrow V[m+1, \ldots, n]$
    $D \leftarrow V[1, \ldots, m]$
    ▷ multiply recursively; don't yet append zeros
    $P_1 \leftarrow$ MULTIPLY$(A, C)$
    $P_2 \leftarrow$ MULTIPLY$(B, D)$
    ▷ here's the tricky part
    $P_3 \leftarrow$ MULTIPLY(ADD$(A, B)$, ADD$(C, D)$)
    ▷ $P_3$ is too large, subtract off $ac$, $bd$
    $P_3 \leftarrow$ SUBTRACT$(P_3, P_1)$
    $P_3 \leftarrow$ SUBTRACT$(P_3, P_2)$
    ▷ combine the answers
    $P \leftarrow$ ADD$(0^{2m} \,||\, P_1, P_2, 0^m \,||\, P_3)$
    return $P$
```

We now prove correctness. The correctness of the base case is clear. Assume now that our inputs are of length n, and the algorithm is correct for all inputs of length less than n. It is easy to see that the recursive calls to MULTIPLY are always done on equal-length arrays. The values represented by P_1, P_2, P_3 are ac, bc, and $ad + bc$ (respectively) by simple algebra, the correctness of SUBTRACT, and the assumed correctness of MULTIPLY on smaller inputs. Note also that if X is an array representing a number x, then $0^i \,||\, A$ is an array representing the number $x \cdot 2^i$. Therefore by the algebra in the problem statement, the algorithm returns the right answer. By induction, it is correct on all inputs.

Problem 2-2. Deterministic versus randomized searching.

(a) Worst-case running time with **one**: n elements examined (if x is at the end of A). With **none**: n elements examined. With **half**: $\lfloor n/2 \rfloor + 1$ elements examined (if all the x's are at the end of A).

(b) Average-case running time with **one**: $(n-1)/2$ (x appears is each position with probability $1/n$, so the average number of elements examined is $(1 + 2 + \cdots + n)/n =$

$(n-1)/2$). With **none**: n elements examined (the entire array must be searched, no matter what). With **half**: the $n/2$ x's are distributed randomly among the n elements, and we want to find the expected value of the first index i such that $A[i] = x$. We can compare our distribution to a Bernoulli distribution: with probability $1/2$, $A[1] = x$. Also, if $A[i] \neq x$ for all $i < j$, then $\Pr[A[j] = x] \geq 1/2$. Under the Bernoulli distribution (where each probability is exactly 1/2), the expected number of elements examined is 2. Since our distribution has even more mass at the lower values, our expectation is slightly less than 2.

(c) The answers for the randomized algorithm are exactly those of the average-case analysis of the deterministic algorithm, but we need not make any assumption about the input (because randomly permuting the array ensures that all inputs to the deterministic subroutine are equally likely). That is, with **one**: $(n-1)/2$ examinations. With **none**: n examinations. With **half**: less than 2 examinations.

Problem 2-3. Lower bounds for merging sorted lists.

(a) There are $\binom{2n}{n}$ ways: we can choose any n elements for one list, and the remaining elements go in the other list. Each different choice of n elements yields a unique pair of lists.

(b) Consider an algorithm that correctly merges two sorted sub-lists A and B, each of length n, into a sorted list C of length $2n$. This algorithm determines the placement of A's elements among C's, and every placement is induced by some input. Therefore the algorithm distinguishes among at least $\binom{2n}{n}$ possibilities, so its decision tree has at least this many leaves. We can lower-bound $\binom{2n}{n}$ in the following way: the sum over i of the binomial coefficients $\binom{2n}{i}$ is 2^{2n}; $\binom{2n}{n}$ is the largest of all $2n+1$ of them, so it is at least $2^{2n}/(2n+1)$. (Stirling's approximation can also give a tighter lower bound.) Therefore some root-to-leaf path in the decision tree involves at least $\lg 2^{2n} - \lg(2n+1) = 2n - o(n)$ comparisons.

(c) Suppose there are two elements a and b from different lists, and the correct sorted output is either $\langle c_1, \ldots, c_j, a, b, c_{j+1}, \ldots, c_k \rangle$ or $\langle c_1, \ldots, c_j, b, a, c_{j+1}, \ldots, c_k \rangle$. For each c_i, in either case every comparison between a and c_i returns the same answer (specifially, $a > c_i$ for $i \leq j$ and $a < c_i$ for $i \geq j+1$). The same goes for b and each c_i. Therefore in order to determine the correct sorted output, the algorithm must compare a and b.

(d) Suppose the two sorted sub-lists are $A = \langle a_1, \ldots a_n \rangle$ and $B = \langle b_1, \ldots, b_n \rangle$. Then if the proper sorted order is $\langle a_1, b_1, a_2, b_2, \ldots, a_n, b_n \rangle$, there are $2n-1$ pairs of elements that are adjacent in the sorted list and come from different sub-lists (specifically, they are a_i, b_i for each $i = 1, \ldots, n$ and b_i, a_{i+1} for each $i = 1, \ldots, n-1$). By the previous part, in order to determine that this is the proper sorted order, any algorithm must perform at least $2n-1$ comparisons.

Problem 2-4. In-place COUNTING-SORT.

(a) Let us first assume the loop invariant, which we will later prove true: each element $A[j]$ for $j > r$ is either in its final sorted position or needs to move left. When the loop exits, $r = 0$, so *every* element of A is either in its final sorted position or needs to move left. However, if the array were unsorted, then *some* element would need to move right. Therefore A is sorted.

Now we prove the loop invariant. It is trivially true upon loop entry, because $r = \mathit{length}[A]$. Now suppose it is true at the beginning of some iteration of the loop. If $j < r$, then either the element $A[r]$ needs to move left, or it is in its final position and was placed there by a previous iteration. If $j \geq r$, then every element that is moved goes to its final sorted position, and in particular, the element the finally ends up at $A[r]$ is in its sorted position. Therefore, before r is decremented, every element $A[j]$ for $j \geq r$ is either in its final sorted position or needs to move left. Then r is decremented and the next loop iteration begins, so the loop invariant is true by induction.

(b) The procedure runs in $O(n)$ time. To see this, notice that whenever an element is moved, it is put in its final sorted position and is never moved again. This is because $C[a]$ (or $C[b]$) is decremented every time an element a (or b) is to be moved. Therefore the total time taken by the inner loop, over all iterations of the outer loop, is $O(n)$.

(c) This is not an acceptable subroutine for RADIX-SORT, because it is not stable. For example, suppose the input array $A = \langle 3, 2, 1, 1 \rangle$. This modified algorithm swaps the relative positions of the 1 entries.

Problem 2-5. Ben Bitdiddle's k-ary heaps.

(a) A k-ary heap can be represented in a 1-dimensional array as follows. The root is kept in $A[1]$, its k children are kept in order in $A[2]$ through $A[k+1]$, their children are kept in order in $A[k+2]$ through $A[k^2+k+1]$, and so on. The two procedures that map a node with index i to its parent and to its jth child (for $1 \leq j \leq k$), respectively, are;

```
K-ARY-PARENT$(i)$
    return $\lfloor (i-2)/k + 1 \rfloor$

K-ARY-CHILD$(i, j)$
    return $k(i-1) + j + 1$
```

To convince yourself that these procedures really work, verify that

$$\text{K-ARY-PARENT}(\text{K-ARY-CHILD}(i, j)) = i \; ,$$

for any $1 \leq j \leq k$. Also notice that the binary heap procedures are a special case of the above procedures when $k = 2$.

(b) Ben is correct when he argues that for $k = 1$ the only operation required in the heapsort algorithm is BUILD-HEAP. Since each node in the heap has only one child, the heap will be comprised of a single root-to-leaf chain containing the sorted elements. Since our array representation of a heap guarantees that a parent will always precede its children in the array, the elements in the array representing this heap will be correctly sorted.

Ben's argument fails when he claims that this BUILD-HEAP operation will run in $O(n)$ time. Look at the code for BUILD-HEAP:

```
BUILD-HEAP($A$)
1  heap-size[$A$] ← length[$A$]
2  for $i$ ← ⌈length[$a$]/$k - 1$⌉ downto $1$
3        do HEAPIFY($A, i$)
```

Since the elements in the subarray $A[\lceil n/k \rceil \ldots n]$ are leaves in the tree, BUILD-HEAP goes through the remaining positions and runs HEAPIFY on each one. Before each call to HEAPIFY(A, i), the subarray $A[i + 1 \ldots n]$ is sorted. Since the unary heap has only one path from the root down to the leaf, the next call to HEAPIFY simply walks along the subarray $A[i + 1 \ldots n]$ and inserts $A[i]$ so that the subarray $A[i \ldots n]$ is now sorted. This is exactly the operation of INSERTION-SORT (but from back-to-front, instead of front-to-back) which has worst case running time $\Theta(n^2)$.

(c) To find the value of k that minimizes this expression, notice that $nk \log_k n = nk \ln n / \ln k$. The derivative of $\frac{k}{\ln k}$ with respect to k is $(\ln k)^{-1} - (\ln k)^{-2}$. Setting this to zero yields $k = e = 2.718...$ as the minimizing value. (In reality, k must be an integer, and it is easy to verify that $k = 3$ minimizes the number of comparisons, subject to this integrality constraint.)