CS 161 Summer 2009
Homework #2 Sample Solutions

Regrade Policy: If you believe an error has been made in the grading of your homework, you may resubmit it for a regrade. If the error consists of more than an error in addition of points, please include with your homework a detailed explanation of which problems you think you deserve more points on and why. We reserve the right to regrade your entire homework, so your final grade may either increase or decrease.

Problem 1 [24 points]

Remarks: (1) You need to specify the constant ϵ and c in formal proof; (2) To apply case 3 of Master Theorem, you need to check regularity condition first.

(a) (3 points) $T(n) = 3T(n/2) + n \log n$

Answer: $T(n) = \Theta(n^{\log_2 3})$.
We will apply case 1 of the Master Theorem. Since $f(n) = n \log n$ and $n^{\log_b a} = n^{\log_2 3} \approx n^{1.58}$. We can pick $\epsilon = 0.1$ to satisfy the conditions of the theorem. Therefore $T(n) = \Theta(n^{\log_2 3})$.

(b) (3 points) $T(n) = 5T(n/5) + n/\log n$

Answer: $T(n) = \Theta(n \log \log n)$.
We have $a = 5$, $b = 5$, $f(n) = n/\log n$. So $n^{\log_b a} = n$. None of the Master Theorem cases may be applied here, since $f(n)$ is neither polynomially bigger or smaller than n, and is not equal to $\Theta(n \log^k n)$ for any $k \geq 0$. Therefore, we will solve this problem by algebraic substitution.

$$\begin{aligned} T(n) &= 5T(n/5) + n/\log n \\ &= 5(5T(n/25) + (n/5)/\log(n/5)) + n/\log n \\ &= 25T(n/25) + n/\log(n/5) + n/\log n \\ &\cdots \\ &= 5^i T(n/5^i) + \sum_{j=0}^{i-1} (n/\log(n/5^j)) \end{aligned}$$

When $i = \log_5 n$, then first term reduces to $5^{\log_5 n}T(1)$, so we have

$$\begin{aligned} T(n) &= n\Theta(1) + \sum_{j=0}^{\log_5 n-1} (n/\log(n/5^j)) \\ &= \Theta(n) + n \sum_{j=0}^{\log_5 n-1} (1/(\log n - j\log 5)) \\ &= \Theta(n) + n/\log 5 \sum_{j=0}^{\log_5 n-1} (1/(\log_5 n - j)) \\ &= \Theta(n) + n\log_5 2 \sum_{k=1}^{\log_5 n} (1/k) \end{aligned}$$

This is the harmonic sum, so we have $T(n) = \Theta(n) + c_2 n \ln(\log_5 n) + \Theta(1) = \Theta(n \log\log n)$

(c) (3 points) $T(n) = 4T(n/2) + n^2\sqrt{n}$

Answer: $T(n) = \Theta(n^2\sqrt{n})$.
We will use case 3 of the Master Theorem, Since $f(n) = n^2\sqrt{n} = n^{2.5}$ and $n^{\log_b a} = n^{\log_2 4} = n^2$. We can pick $\epsilon = 0.1$ to satisfy the conditions of the theorem. Moreover if $c = 0.9$ we can verify that $4(n/2)^{2.5} \leq c\ n^{2.5}$. Therefore the premises for case 3 hold and we conclude that $T(n) = \Theta(n^2\sqrt{n})$.

(d) (3 points) $T(n) = 3T(n/3 + 5) + n/2$

Answer: $T(n) = \Theta(n \log n)$.
We will solve the problem by guess-and-check. By induction, $T(n)$ is a monotonically increasing function. Thus, for the lower bound, since $T(n/3) \leq T(n/3 + 5)$, using case 2 of the Master Theorem, we have $T(n) = \Omega(n \log n)$.
For the upper bound, we can show $T(n) = O(n \log n)$.
For the base cases $\forall\ n \leq 30$, we can choose a sufficiently large constant d_1 such that $T(n) < d_1 n \log n$.
For the inductive step, assume for all $k < n$, that $T(n) < d_1 n \log n$. Then for $k = n > 30$, we have $n/3 + 5 < n - 15$, then

$$\begin{aligned} T(n) &= 3T(n/3 + 5) + n/2 \\ &< 3T(n - 15) + n/2 \\ &< 3(3T((n - 15)/3 + 5) + (n - 15)/2) + n/2 \\ &< 9T(n/3) + 3(n - 15)/2 + n/2 \\ &< 9d_1(n/3)\log(n/3) + 2n - 45/2 \\ &< 3d_1 n \log(n/3) + 2n - 45/2 \\ &< 3d_1 n \log n \end{aligned}$$

The last line holds for $n \geq 1$ and $d_1 > \frac{2}{3\log 3}$. Thus we can proof by induction that $T(n) < cn\log n$ for all n, where $c = max\{d_1, 3d_1, \frac{2}{\log 3}\}$.
Therefore, $T(n) = O(n\log n)$ and $T(n) = \Theta(n\log n)$.

(e) (3 points) $T(n) = 2T(n/2) + n/\log n$

Answer: $T(n) = \Theta(n\log\log n)$.
Similar to part(b).

(f) (3 points) $T(n) = T(n/2) + T(n/4) + T(n/8) + n$

Answer: $T(n) = \Theta(n)$.
We solve this problem by guess-and-check. The total size on each level of the recurrence tree is less than n, so we guess that $f(n) = n$ will dominate.
Base case, for $n_0 = 1$, we have $T(n) = n$.
Inductive step: Assume for all $i < n$ that $c_1 n \leq T(i) \leq c_2 n$. Then,

$$\begin{aligned} c_1 n/2 + c_1 n/4 + c_1 n/8 + kn &\leq T(n) \leq c_2 n/2 + c_2 n/4 + c_2 n/8 + kn \\ c_1 n(7/8 + k/c_1) &\leq T(n) \leq c_2 n(7/8 + k/c_2) \end{aligned}$$

If $c_1 \geq 8k$ and $c_2 \leq 8k$, then $c_1 n \leq T(n) \leq c_2 n$. So, $T(n) = \Theta(n)$.

(g) (3 points) $T(n) = T(n-1) + 1/n$

Answer: $T(n) = \Theta(\log n)$.
We solve this problem by algebraic substitution.

$$\begin{aligned} T(n) &= T(n-1) + 1/n \\ &= T(n-2) + 1/(n-1) + 1/n \\ &\cdots \\ &= \Theta(1) + \sum_{i=1}^{n} 1/i \\ &= \Theta(1) + \ln n \end{aligned}$$

(j) (3 points) $T(n) = \sqrt{n}T(\sqrt{n}) + n$

Answer: $T(n) = \Theta(n\log\log n)$.
We solve it by algebraic substitution.

$$\begin{aligned} T(n) &= \sqrt{n}T(\sqrt{n}) + n \\ &= n^{1/2}(n^{1/4}T(n^{1/4} + n^{1/2})) + n \\ &= n^{3/4}T(n^{1/4}) + 2n \\ &= n^{3/4}(n^{1/8}T(n^{1/8} + n^{1/4})) + 2n \\ &= n^{7/8}T(n^{1/8}) + 3n \\ &\cdots \\ &= n^{1-1/2^k}T(n^{1/2^k}) + kn \end{aligned}$$

When $n^{1/2^k}$ falls under 2, we have $k > \log\log n$. We then have $T(n) = n^{1-1/\log n}T(2)+ n\log\log n = \Theta(n\log\log n)$.

Problem 2 [5 points]

Answer: $a = 48$.
$A : T(n) = 7T(n/2) + n^2$. We have $a = 7$, $b = 2$, $f(n) = n^2$. Apply case 1 of the Master Theorem, we get $T(n) = \Theta(n^{\log_2 7})$.
$A' : T'(n) = aT'(n/4) + n^2$. We have $a = a$, $b = 4$, $f(n) = n^2$. If $\log_2 7 > \log_4 a > 2$, Apply case 1 of the Master Theorem, we will get $T'(n) = \Theta(n^{log_4 a}) < \Theta(n^{log_2 7}) = T(n)$. Solving the inequation, we get $a < 49$.
When $\log_4 a \leq 2$, by case 2 and case 3 of the Master Theorem, we won't get a faster algorithm than A.

Problem 3 [30 points]

(a) (10 points)

Insertion-Sort: $O(kn)$. Define an "inversion" in A as a pair (i, j) where $i < j$ and $A[i] > A[j]$. If every element is within k slots from its proper position, then there are at most $O(kn)$ inversions. Thus the running time for INSERTION-SORT is $O(n + I) = O(kn)$, where I is the number of inversions in A.
The reason is because I is exactly number of shifting operations we need in INSERTION-SORT. For the inner loop of the algorithm, each element $A[i]$ is shifted left as many times as l_i where l_i is number of elements j so that $j < i$ and $A[j] > A[i]$. Therefore

$$\begin{aligned} \textit{number of shifting operations needed} &= \sum_{i=1}^{n} l_i \\ &= \sum_{i=1}^{n} |\{(j,i)|(j,i) \textit{ is an inversion in } A, \forall j\}| \\ &= \textit{total number of inversions in } A \end{aligned}$$

Now consider the outer loop of the INSERTION-SORT algorithm. It iterates n times as the index traverses the array and it is independent of the number of inversions. Therefore the total running time of INSERTION-SORT is $O(I + n) = O(kn)$.

Merge-Sort: $\Theta(n\log n)$. Same Recurrence function $T(n) = 2T(n/2) + \Theta(n)$.

QuickSort:
Best-case scenario, rewrite recurrence as $T(n) = T(k) + T(n-k) + \Theta(n)$. The depth

of the recurrence tree is n/k. Thus $T(n) = O(n^2/k)$. When $k << n$, it is close to $O(n^2)$.
Worst-case scenario (sorted), rewrite recurrence as $T(n) = T(1) + T(n-1) + \Theta(n)$. Thus $T(n) = O(n^2)$.

(b) (5 points) $O(kn)$. The idea of the BUBBLE-SORT algorithm is, starting from the left, compare adjacent items and keep "bubbling" the larger one to the right until all items reach their proper positions.
From $3(a)$, we know there are at most $O(kn)$ inversions. After each swap operation in line 6, the number of inversion will decrease by one. Thus line 6 will be executed at most $O(kn)$ times, which is the bottle-neck of the running time.

(c) (10 points) For the solution to this problem, we are going to use a heap. We assume that we have a heap with the following subroutines:

- MAKE-HEAP() returns a new empty heap.
- INSERT($H, key, value$) inserts the key/value pair into the heap.
- EXTRACT-MIN(H) removes the key/value pair with the smallest key from the heap, returning the value.

We consider A as a set of sorted lists. In particular, notice that $A[i] < A[i+2k+1]$, for all $i \leq n-2k-1$. That's because the element at slot i can at most move forwards during sorting to slot $i+k$ and the element at slot $i+2k+1$ can at most move backwards during sorting to slot $i+k+1$. For simplicity, assume n is divided by $2k$. We divide the array A into $2k$ lists defined as follows:

$$\begin{aligned} A_1 &= [A[1], A[2k+1], A[4k+1], \ldots, A[n-(2k-1)]] \\ A_2 &= [A[2], A[2k+2], A[4k+2], \ldots, A[n-(2k-2)]] \\ &\ldots \\ A_{2k} &= [A[2k], A[4k], A[6k], \ldots, A[n]] \end{aligned}$$

Each of these lists is sorted, and of size $\leq n$. We can sort these lists in $O(n \log k)$ time using the SORT-K-SORTED algorithm below.

Algorithm 1 SORT-K-SORTED(A, n, k)

```
1: H = MAKE-HEAP()
2: for i = 1 to 2k do
3:    INSERT(H, A[i], i)
4: end for
5: for i = 1 to n do
6:    j=EXTRACT-MIN(H)
7:    B[i] = A[j]
8:    if j + 2k ≤ n then
9:       INSERT(H, A[j + 2k], j)
10:   end if
11: end for
12: return B
```

Running time analysis: Since there are never more than $2k$ element in the heap, Each EXTRACT-MIN and INSERT operation requires $O(\log k)$ time. The second loop is is repeated n times, resulting in an $O(n \log k)$ running time.

(d) (5 points) The key invariant is to show that after every iteration of the loop, the heap contains the smallest element in every list. A formal induction proof is required here.

Alternative Solution: Recall we already know how to merge two sorted lists that contain n elements in O(n) time. It is possible, then to merge the lists in a *tournament*. For example, for $2k = 8$, where $A + B$ means to merge lists A and B:
Round 1: $(A_1 + A_2), (A_3 + A_4), (A_5 + A_6), (A_7 + A_8)$
Round 2: $(A_1 + A_2 + A_3 + A_4), (A_5 + A_6 + A_7 + A_8)$
Round 3: $(A_1 + A_2 + A_3 + A_4 + A_5 + A_6 + A_7 + A_8)$
Notice that there are $\log k$ merge steps, each of which merges n elements and hence has a cost of $O(n)$. This leads to the total running time of $O(n \log k)$.

Problem 4 [15 points]

(a) (5 points) MERGE-SORT is the most amenable one because you can give a machine a part of the array in any order. QUICKSORT and INSERTION-SORT are bad because one iteration depends on the state of the array based on previous iterations, which is complicated to communicate between machines.

(b) (5 points) Split the array into $m = 2^i$ subarrays. Call MERGE-SORT on each subarray on separate machine. Once machines start to returning answers, use free machines to merge subarrys that are adjacent to each other.

(c) (5 points) Running time: $O((n/m)\log(n/m) + n)$.
Since we are splitting the array into $m = 2^i$ parts, each call of MERGE-SORT on a subarray of size n/m will take $O((n/m)\log(n/m))$.Then we need to merge m sorted array into a big one, which will take linear time $O(n)$.
Thus the total time is $O((n/m)\log(n/m) + n)$.

Alternative Solution: At the Merge step, we can use $m/2$ machines to merge all pairs of sorted subarrays, then $m/4$ machines for the next merge step, etc. This will give the same Big-O running time.

Problem 5 [8 points]

Insertion-Sort: Stable. We can prove it using the following loop invariant: At the start of each interaction of the **for** loop of lines 1-8 (page 17 of CLRS), if $A[a] = A[b], a < b \leq j-1$ and distinct, then $A[a]$ appeared before $A[b]$ in teh initial array. Proof by induction is omitted here.

Merge-Sort: Stable. We can prove it by induction on the fact that MERGE-SORT (page 29 and 32 on CLRS) is stable.
Base case: When we call MERGE-SORT with indices p and r such that $p \not< r$ (therefore $p == r$), we return the same array. So calling MERGE-SORT on an array of size one returns the same array, which is stable.
Induction: We assume that calling MERGE-SORT on an array of size less than n returns a stably sorted array. We then show that if we call MERGE-SORT on an array of size n, we also return a stably sorted array. Each call of MERGE-SORT contains two calls to MERGE-SORT on half arrays, and in addition, merges these two subarrays and returns. Since we assume that the calls to MERGE-SORT on smaller arrys return stably sorted arrays, we only need to show that the MERGE step on two stably sorted arrays returns a stable array. if $A[i] = A[j]$, $i < j$ in the initial array, we need $f(i) < f(j)$ in the new array, where f is the function which gives the new positions in the sorted array. If i and j are in the same half of the recursive MERGE-SORT call, then by assumption, it hold. If i and j are in different subarrays, then we know that i is in the left subarray and j is in the right subarray since $i < j$. In the MERGE step, we take the elements from the left subarray while it is less than or equal to the right subarray element (line 13). Therefore, we will take element i before taking element j and $f(i) < f(j)$, the claim we are trying to prove.

Bubble-Sort: Stable. Formal proof by induction is omitted here. The key point is that equal value elements are not swapped (line 5 in the pseudocode in Problem 3(b)).

QuickSort: NOT stable. To make it stable, we can add a new field to each array element, which indicates the index of that element in the original array. Then, when we

sort, we can sort on the condition (line 4 on page 146 of CLRS) $A[j] > x$ OR $A[j] == x$ AND $index(A[j]) < index(x)$. At the end of the sort, we are guaranteed to have all the elements in sorted order, and for any $i < j$, such that $A[i]$ equals $A[j]$, we will have $index(A[j]) < index(A[i])$ so the sort will be stable.

Extra Credit [5 points]

We will give a proof using the decision-tree technique.

We already know that sorting the array requires $\Omega(n \log n)$ comparisons. If $k > n/2$, then $n \log n = \Omega(n \log k)$, and the proof is complete. For the remainder of this proof, assume that $k \leq n/2$.

Our goal is to provide a lower-bound on the number of leaves in a decision tree for an algorithm that sorts an array in which every element is within k slots of its proper position. We therefore provide a lower bound on the number of possible permutations that satisfy this condition.

First, break the array of size n into $\lfloor n/k \rfloor$ blocks, each of size k, and the remainder of size $n \pmod k$. For each block, there exist $k!$ permutations, resulting in at least $(k!)^{\lfloor n/k \rfloor}$ total permutations of the entire array. None of these permutations move an element more than k slots.

Notice that this undercounts the total number of permutations, since no element moves from one k-element block to another, and we ignore permutations of elements in the remainder block.

We therefore conclude that the decision tree has at least $(k!)^{\lfloor n/k \rfloor}$ leaves. Since the decision tree is a binary tree, we can then conclude that the height of the decision tree is

$$\begin{aligned}
&\geq \log((k!)^{\lfloor n/k \rfloor}) \\
&\geq \lfloor n/k \rfloor \log(k!) \\
&\geq \lfloor n/k \rfloor c_1 k \log k \\
&\geq c_1 (n-k) \log k \\
&\geq c_1 n \log k/2 \\
&= \Omega(n \log k)
\end{aligned}$$

The last step follows because of our assumption that $k \leq n/2$.