CS161: Design and Analysis of Algorithms **Summer 2004**

Problem Set #3 Solutions

General Notes

- **Regrade Policy:** If you believe an error has been made in the grading of your problem set, you may resubmit it for a regrade. If the error consists of more than an error in addition of points, please include with your problem set a detailed explanation of which problems you think you deserve more points on and why. We reserve the right to regrade your entire problem set, so your final grade may either increase or decrease.
- If we ask for an answer in terms of n and d, please give the answer in terms of n and d and do not assume that d is a constant. Many people did this on the first problem as well as the skip list problem from the last problem set.
- In part (b) of problem 3, many people only showed that the probability that **any** slot had k elements was $\leq nQ_k$. You had to show the **maximum** number of elements was k.

1. **[14 points]** Analysis of d-ary Heaps

   Throughout this problem, when asked to give an implementation of a certain operation, please provide pseudocode or a verbal description with the same level of clarity and detail. When asked to analyze running times, your analysis does not need to be formal, but should be justified.

   Note that we are asking you to solve 6-2(e) before 6-2(d). Keep in mind that you may always (not only in this problem) use the results of any previous parts in answering a later part of a multi-part problem.

   (a) **[1 point]** Do problem 6-2(a) on page 143 of CLRS.

   Answer: A d-ary heap can be represented in a 1-dimensional array by keeping the root of the heap in $A[1]$, its d children in order in $A[2]$ through $A[d+1]$, their children in order in $A[d+2]$ through $A[d^2+d+1]$, and so on. The two procedures that map a node with index i to its parent and to its j^{th} child (for $1 \leq j \leq d$) are

   ```
   D-PARENT(i)
   1    return ⌈(i − 1)/d⌉

   D-CHILD(i, j)
   2    return d(i − 1) + j + 1
   ```

   (b) **[1 point]** Do problem 6-2(b) on page 143 of CLRS.

   Answer: Since each node has d children, the total number of nodes in a tree of height h is bounded above by $1 + d + \ldots + d^h$ inclusively and below by $1+d+\ldots+d^{h-1}$ exclusively. This yields $h = \lceil \log_d(1+n(d-1)) \rceil - 1 = \Theta(\log_d n)$.

(c) **[2 points]** Give an efficient implementation of HEAPIFY in a d-ary max-heap. Analyze its running time in terms of d and n.

Answer: First, we will need to find if the root element given to D-HEAPIFY is larger than all its children. If not, we swap it with its largest child and recursively call D-HEAPIFY on that child.

```
D-HEAPIFY(A, i, d)
1   largest ← i
2   for j ← 1 to d
3       j ← D-CHILD(i, j)
4       if j ≤ heapsize[A] and A[j] > A[largest]
5           then largest ← j
6   if largest ≠ i
7       then exchange A[i] ↔ A[largest]
8           D-HEAPIFY(A, largest, d)
```

This algorithm does $\Theta(d)$ comparisons in each call to D-HEAPIFY as well as $O(\log_d n)$ calls to HEAPIFY, one for each level of the tree. Therefore, the total running time for this algorithm is $O(d \log_d n)$.

(d) **[5 points]** Give an efficient implementation of BUILD-HEAP in a d-ary max-heap. Analyze its running time in terms of d and n.

Answer: To build a heap, we can simply assume that our array is already a heap and call D-HEAPIFY in a bottom up manner to convert the array into a d-max-heap. The leaves are already heaps since they have no children, so we start with the lowest non-leaf element. By our procedure for D-PARENT, the parent of the last element is $\lceil (n-1)/d \rceil$. By the time we call D-HEAPIFY on any element, D-HEAPIFY will have already been called on all of its children and therefore the call is valid since the children are all themselves heaps.

```
D-BUILD-HEAP(A, d)
1   heapsize[A] ← length[A]
2   for i ← ⌈(length[A] − 1)/d⌉ downto 1
3       do D-HEAPIFY(A, i, d)
```

We can bound the running time of this algorithm by modifying the analysis in section 6.3 of CLRS. Notice that there are

$$d^{\log_d(n(d-1))-(h+1)} = d^{\log_d\left(\frac{n(d-1)}{d^{h+1}}\right)} = \frac{n(d-1)}{d^{h+1}}$$

nodes at each height h. From part (c), the time for each call to HEAPIFY at a node of height h is $O(dh)$. So, the running time

$$
\begin{aligned}
T(n) &\leq \sum_{h=0}^{\lg_d(n(d-1))} dh\frac{n(d-1)}{d^{h+1}} \\
&< n(d-1)\sum_{h=0}^{\infty}\frac{h}{d^h} \\
&\leq n(d-1)\frac{1/d}{(1-1/d)^2} \qquad \text{A.8} \\
&= n\frac{d}{d-1} \\
&= n\left(1+\frac{1}{d-1}\right)
\end{aligned}
$$

For any $d \geq 2$ we have $1 \leq 1 + 1/(d-1) \leq 2$, so $T(n) = \Theta(n)$.

(e) **[2 points]** Do problem 6-2(c) on page 143 of CLRS.

Answer: The procedure D-EXTRACT-MAX is the same as for the binary heap. We return the value of $A[1]$, move the element in $A[heapsize]$ into $A[1]$, decrement $heapsize$, and then call D-HEAPIFY on the root of the heap.

```
D-EXTRACT-MAX(A, d)
1   if heapsize[A] < 1
2      then error
3   max ← A[1]
4   A[1] ← A[heapsize]
5   heapsize ← heapsize − 1
6   D-HEAPIFY(A, 1, d)
7   return max
```

All operations besides D-HEAPIFY take $O(1)$ time. D-HEAPIFY takes time $\Theta(d \log_d n)$ so D-EXTRACT-MAX takes time $\Theta(d \log_d n)$.

(f) **[2 points]** Do problem 6-2(e) on page 143 of CLRS.

Answer: If $k < A[i]$, then D-INCREASE-KEY does not change the heap. Otherwise, it will set $A[i]$ to k and move the element upwards toward the root until it is smaller than its parent.

```
D-INCREASE-KEY(A, i, k, d)
1   if k > A[i]
2      then A[i] ← k
3         while i > 1 and A[D-PARENT(A, i, d)] < A[i]
4            do exchange A[i] ↔ A[D-PARENT(A, i, d)]
5               i ← A[D-PARENT(A, i, d)]
```

The running time of the algorithm is proportional to the height of the heap, and each level takes a constant amount of work, so the overall time is $\Theta(\log_d n)$.

(g) **[1 point]** Do problem 6-2(d) on page 143 of CLRS.

Answer: The procedure D-INSERT is the same as for the binary heap. We add the element to the end of the array and set its key to $-\infty$. Then, we call D-INCREASE-KEY to increase to increase the key to key.

D-INCREASE-KEY(A, key, d)
1 $\quad heapsize[A] \leftarrow heapsize[A] + 1$
2 $\quad A[heapsize[A]] \leftarrow -\infty$
3 $\quad$ D-INCREASE-KEY$(A, heapsize[A], key, d)$

The first two operations take constant time, and the call to D-INCREASE-KEY takes time $\Theta(\log_d n)$. So, the running time for D-INCREASE-KEY is $\Theta(\log_d n)$.

2. **[23 points]** Average-Case Lower Bounds on Comparison Sorting

Throughout the next two problems, when asked to prove, argue, show or conclude something, please make your explanations as clear and complete as possible.

(a) **[4 points]** Do problem 8-1(a) on page 178 of CLRS.

Answer: Since the sort can be run on any input, the algorithm must be able to reach any permutation of the input in order to get the correct sorted order as an answer. Because A is deterministic, any input will always follow exactly one path in T_A. There are $n!$ possible permutations of n elements, and each permutation should lead to a distinct leaf in T_A in order to yield the correct sorted order. Since we assume that every permutation of A's input is equally likely and since the probabilities must sum to 1, each leaf must have probability $1/n!$. Any other leaf is never reached and thus must have probability 0, as needed to be shown.

(b) **[2 points]** Do problem 8-1(b) on page 178 of CLRS.

Answer: The external path length T is given by the sum of the depths of all the leaves contained in the right subtree plus the sum of the depths of all the leaves contained in the left subtree. The sum of the depths of the leaves in the right subtree is the sum of the depths up to the root of the right subtree plus one for each leaf in the right subtree, and likewise for the sum of the depths in the left subtree. So, the total depth $D(T) = D(RT)+$ number of leaves in $RT + D(LT)$ + number of leaves in LT. Each leaf must be in exactly one of the left or right subtrees, so $D(T) = D(RT) + D(LT)+$ number of leaves $= D(RT) + D(LT) + k$.

(c) **[3 points]** Do problem 8-1(c) on page 178 of CLRS.

Answer: Consider a decision tree T with k leaves that achieves the minimum. Let i_0 be the number of leaves in LT and $k - i_0$ be the number of leaves in RT. From part (b), we have $D(T) = D(LT) + D(RT) + k$. Notice that this implies that LT and RT also have the minimum external path length over all subtrees

with i_0 and $k - i_0$ leaves respectively, since otherwise we could replace them with better trees and get a lower value of $D(T)$.

Finally, the optimal tree T represents the best possible split of the k leaves into (optimal) subtrees with i_0 and $k - i_0$ leaves. Since each subtree must have at least one leaf, we have $d(k) = \min_{1 \le i \le k-1}\{d(i) + d(k-i) + k\}$.

(d) **[4 points]** Do problem 8-1(d) on page 178 of CLRS.

Answer: We can find the minimum by setting the derivative with respect to i equal to zero.

$$\begin{aligned}
\frac{d}{di}(i \lg i + (k-i)\lg(k-i)) &= (i)(1/i)(\lg e) + (1)(\lg i) + \\
&\quad (k-i)(1/(k-i))(-\lg e) + (-1)(\lg(k-i)) \\
&= \lg e + \lg i - \lg e - \lg(k-i) \\
&= \lg i - \lg(k-i)
\end{aligned}$$

This is zero when $i = k - i$ or $i = k/2$. We can verify that this is in fact a minimum by taking the second derivative and checking that it is positive.

We can now check that $d(k) = \Omega(k \lg k)$ using the guess-and-check method. Assume that $d(i) \ge ci \lg i$ for $i < k$.

$$\begin{aligned}
d(k) &= \min_{1 \le i \le k-1}(d(i) + d(k-i) + k) \\
&\ge \min_{1 \le i \le k-1}(ci \lg i + c(k-i)\lg(k-i) + k) \\
&= k + c \min_{1 \le i \le k-1}\{i \lg i + (k-i)\lg(k-i)\} \\
&\ge k + c((k/2)\lg(k/2) + (k/2)\lg(k/2)) \\
&= ck \lg(k/2) + k \\
&= ck(\lg k - \lg 2) + k \\
&= ck \lg k + k(1-c) \\
&\ge ck \lg k
\end{aligned}$$

with the last inequality holding as long as $c \le 1$. Therefore, $d(k) = \Omega(k \lg k)$.

(e) **[2 points]** Do problem 8-1(e) on page 178 of CLRS.

Answer: In part (a), we showed that any deterministic tree T_A has at least $n!$ leaves. Since $d(k)$ is the minimum value over all decision trees with k leaves, we know that $D(T_A) \ge d(k)$. In part (d), we showed that $d(k) = \Omega(k \lg k)$. Since T_A has $k \ge n!$ leaves, we know $D(T_A) \le d(k) = \Omega(k \lg k) = \Omega(n! \lg(n!))$.

The expected time to sort n elements is the sum over all leaves of the length of the path to that leaf times the probability of reaching that leaf. Since the probability of reaching each leaf is $1/n!$ (ignoring leaves with probability 0 that

contribute nothing to the sum), this is precisely $D(T_A)/n!$. Thus, $E[T(n)] = \Omega(n! \lg(n!))/n! = \Omega(\lg n!) = \Omega(n \lg n)$.

(f) **[8 points]** Do problem 8-1(f) on page 178 of CLRS.

Answer: Consider the decision tree T_B for algorithm B. We will work from the bottom to the top, replacing each randomization node with the branch that has the smallest external path length. We call the resulting tree T_A and we show that T_A represents a valid deterministic sorting algorithm and $E[T_A(n)] \leq E[T_B(n)]$.

First, consider the issue of correctness. Since the decision tree for A represents a possible execution of B (with appropriate random choices made at each step), and B is correct, the resulting decision tree must represent a valid sorting algorithm. Since all randomization nodes have been removed, this algorithm is deterministic.

Now notice that at each randomization node N, the expected execution time after that node is reached is equal to $\sum_{i=1}^{r}(1/r)E[T(S_i)]$, where S_i is the i^{th} subtree of N. Clearly we have $E[T(N)] \geq min_{i=1}^{r} E[T(S_i)]$, since all terms in the average must be at least the minimum. Thus, each transformation that replaces a randomization node with its best child does not increase the expected running time.

We formalize these notions slightly. Consider a sequence of decision trees $T_1, \ldots, T_m$ where $T_1 = T_B$ and $T_m = T_A$, and each T_i was obtained from T_{i-1} by choosing a randomization node all of whose descendants are deterministic and replacing it with the subtree with the smallest external path length. Notice that since no descendants of N are randomization nodes, the notion of expected running time and external path length (given random input) is the same for its children.

The correctness argument above applies, as does the running time argument: $E[T(T_i)] \leq E[T(T_{i-1})]$. Finally, when all the randomization nodes are removed, we have a deterministic sorting algorithm with the decision tree $T_m = T_A$ and expected running time no worse than the expected running time of B.

3. **[18 points]** Slot-Size Bound for Chaining

(a) **[2 points]** Do problem 11-2(a) on page 250 of CLRS.

Answer The probability that exactly k keys hash to the same spot is given by first choosing k keys from the n, multiplying by the probability that those k keys hashed to a particular spot, and multiplying by the probability that the remaining $n - k$ keys hashed to a different spot (see appendix $C.4$ for a discussion of the binomial distribution if this is unclear). Thus we have

$$Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k}.$$

(b) **[2 points]** Do problem 11-2(b) on page 250 of CLRS.

Answer: The probability of the event $M = k$ is equal to $Pr\{\cup_{i=1}^{n} E_i\}$, where E_i is the event that slot i has the maximum number of elements and that slot i has exactly k elements.

The probability of a union of a set of events is bounded by the sum of the probabilities of those events (with equality if and only if those events are mutually exclusive), so $Pr\{M = k\} \leq \sum_{i=1}^{n} Pr\{E_i\} = nPr\{E_1\}$ due to symmetry.

Finally, observe that the event E_i subsumes the event that exactly k keys hash to slot i, because E_i represents the event { k keys hash to i **and** i has the maximum number of elements}. Thus, $Pr\{E_i\} \leq Q_k$. Combining this with the previous equation yields

$$P_k = Pr\{M = k\} \leq nPr\{E_1\} \leq nQ_k.$$

(c) **[2 points]** Do problem 11-2(c) on page 250 of CLRS.

Answer:

$$\begin{aligned} Q_k &= \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k} \\ &\leq \left(\frac{1}{n}\right)^k \binom{n}{k} && \text{since } \left(1 - \frac{1}{n}\right)^{n-k} \leq 1 \\ &\leq \left(\frac{1}{n}\right)^k \left(\frac{en}{k}\right)^k && \text{equation C.5} \\ &= \left(\frac{e}{k}\right)^k \end{aligned}$$

(d) **[8 points]** Do problem 11-2(d) on page 250 of CLRS.

Answer: We will find a $c > 1$ such that $(e/k_0)^{k_0} < 1/n^3$. Since $(e/k)^k$ is a decreasing function for $k > e$ (when $e/k < 1$), this means that $Q_k < 1/n^3$ for $k \geq k_0$. Finally, combining this with the result of part (b) will yield the desired bound on P_k.

Thus, we want $(e/k_0)^{k_0} < 1/n^3$. Taking lg of both sides yields

$$\begin{aligned} k_0 \lg\left(\frac{e}{k_0}\right) &< -3 \lg n \\ k_0(\lg e - \lg k_0) &< -3 \lg n \\ \frac{c \lg n}{\lg \lg n}\left(\lg e - \lg \frac{c \lg n}{\lg \lg n}\right) &< -3 \lg n \\ \frac{c}{\lg \lg n}(\lg e - \lg c - \lg \lg n + \lg \lg \lg n) &< -3 \\ \frac{c}{\lg \lg n}(-\lg e + \lg c + \lg \lg n - \lg \lg \lg n) &> 3 \end{aligned}$$

For $c > e$, the $\lg c$ term is larger than the $\lg e$ term, so we need to make sure the rest of the left side is greater than the right side, i.e.

$$\begin{aligned}
\frac{c}{\lg\lg n}(\lg\lg n - \lg\lg\lg n) &> 3 \\
c &> \frac{3\lg\lg n}{\lg\lg n - \lg\lg\lg n} \\
c &> \frac{3}{1 - \frac{\lg\lg\lg n}{\lg\lg n}}
\end{aligned}$$

As n gets large, the second term in the denominator will go to zero; in fact for $n \geq 16$ we have $\frac{\lg\lg\lg n}{\lg\lg n} \leq 1/2$ so the right hand side is ≤ 6.

Thus, we can choose an appropriate constant c to guarantee that $(e/k_0)^{k_0} < 1/n^3$ and therefore $Q_k < 1/n^3$ for $k \geq k_0$. Thus, $P_k \leq nQ_k < 1/n^2$ for $k \geq k_0$.

(e) **[4 points]** Do problem 11-2(e) on page 250 of CLRS.
Answer:

$$\begin{aligned}
E[M] &= \sum_{i=1}^{n} i \cdot Pr\{M = i\} \\
&= \sum_{i=1}^{k_0} i \cdot Pr\{M = i\} + \sum_{i=k_0+1}^{n} i \cdot Pr\{M = i\} \\
&\leq k_0 \sum_{i=0}^{k_0} Pr\{M = i\} + n \sum_{i=k_0+1}^{n} Pr\{M = i\} \\
&= k_0 Pr\{M \leq k_0\} + nPr\{M > k_0\} \\
&= n \cdot Pr\left\{M > \frac{c\lg n}{\lg\lg n}\right\} + \frac{c\lg n}{\lg\lg n} \cdot Pr\left\{M \leq \frac{c\lg n}{\lg\lg n}\right\}
\end{aligned}$$

Plugging in the values from part (d), we have

$$\begin{aligned}
E[M] &\leq (n)(n - k_0)(1/n^2) + \frac{c\lg n}{\lg\lg n} \cdot 1 \\
&\leq (n)(n)(1/n^2) + \frac{c\lg n}{\lg\lg n}
\end{aligned}$$

Thus, $E[M] \leq 1 + c\lg n/\lg\lg n = O(\lg n/\lg\lg n)$.