

CMPS 102 Solutions to Homework 3

Lindsay Brown, lbrown@soe.ucsc.edu

October 13, 2005

Problem 1. 6.3-2 p.135 build-max-heap

BUILD-MAX-HEAP(A)

1. $\text{heap-size}[A] \leftarrow \text{length}[A]$
2. **for** $i \leftarrow \lfloor \text{length}[A]/2 \rfloor$ **downto** 1
3. **do** MAX-HEAPIFY (A, i)

When the procedure MAX-HEAPIFY(A, i) is called it is assumed that the left and right subtrees of node i are max-heaps. When building a max-heap from array A we cannot assume this property holds. So if we begin calling MAX-HEAPIFY on the root of the tree $A[1]$, the left and right subtrees may not be max-heaps. However, if we begin by calling MAX-HEAPIFY on the parents of the leaves, then the partial order property will hold as BUILD-MAX-HEAP moves up the tree.

Problem 2. 6.5-7 p.142 heap-delete

Let A be an n element max-heap. HEAP-DELETE(A, i) deletes the item in node i from heap A .

HEAP-DELETE (A, i)

1. $temp \leftarrow A[i]$
2. $A[i] \leftarrow A[\text{heap-size}[A]]$
3. $\text{heap-size}[A] \leftarrow \text{heap-size}[A] - 1$
4. **if** $A[i] < temp$
5. MAX-HEAPIFY(A, i)
6. **else** HEAP-INCREASE-KEY($A, i, A[i]$)

$A[\text{heap-size}[A]]$ is moved to node i and the heap-size is decremented. Now $A[i]$ may be smaller than its children, or larger than its parent, violating the max-heap property. MAX-HEAPIFY (Cormen, p.130) works in a top-down manner fixing any violations beginning with the subtree rooted at node i . If $A[i]$ is larger than its parent, then we need to fix violations moving up the tree. We use HEAP-INCREASE-KEY (Cormen, p.140) to traverse the path from node i toward the root to find the correct node for the key $A[i]$.

The upper bound on the running time of HEAP-DELETE is simply the upper-

bound of MAX-HEAPIFY ($O(\lg n)$) plus the upper-bound of HEAP-INCREASE-KEY ($O(\lg n)$). For an n element heap the worst case running time of HEAP-DELETE is $O(\lg n) + O(\lg n) = O(\lg n)$.

Problem 3. 7.4-2 p.159 quicksort's best-case running time

To show that quicksort's best-case running time is $\Omega(n \lg n)$, we use a technique similar to the one used in Section 7.4.1 to show that its worst-case running time is $O(n^2)$.

Let $T(n)$ be the best-case time for the procedure QUICKSORT on an input of size of n . We have the recurrence

$$T(n) = \min_{1 \leq q \leq n-1} (T(q) + T(n-q-1)) + \Theta(n).$$

We guess that $T(n) \geq cn \lg n$ for some constant c . Substituting this guess into the recurrence, we obtain

$$\begin{aligned} T(n) &\geq \min_{1 \leq q \leq n-1} (cq \lg q + c(n-q-1) \lg(n-q-1)) + \Theta(n) \\ &= c \min_{1 \leq q \leq n-1} (q \lg q + (n-q-1) \lg(n-q-1)) + \Theta(n). \end{aligned}$$

As we'll show below, the expression $q \lg q + (n-q-1) \lg(n-q-1)$ achieves a minimum over the range $1 \leq q \leq n-1$ when $q = n-q-1$, or $q = (n-1)/2$, since the first derivative of the expression with respect to q is 0 when $q = (n-1)/2$ and the second derivative of the expression is positive. (It doesn't matter that q is not an integer when n is even, since we're just trying to determine the minimum value of a function, knowing that when we constrain q to integer values, the function's value will be no lower.)

Choosing $q = (n-1)/2$ gives us the bound

$$\begin{aligned} &\min_{1 \leq q \leq n-1} (q \lg q + (n-q-1) \lg(n-q-1)) \\ &\geq \frac{n-1}{2} \lg \frac{n-1}{2} + (n - \frac{n-1}{2} - 1) \lg(n - \frac{n-1}{2} - 1) \\ &= (n-1) \lg \frac{n-1}{2}. \end{aligned}$$

Continuing with our bounding of $T(n)$, we obtain, for $n \geq 2$,

$$\begin{aligned} T(n) &\geq c(n-1) \lg \frac{n-1}{2} + \Theta(n) \\ &= c(n-1) \lg(n-1) - c(n-1) + \Theta(n) \\ &= cn \lg(n-1) - c \lg(n-1) - c(n-1) + \Theta(n) \\ &\geq cn \lg(n/2) - c \lg(n-1) - c(n-1) + \Theta(n) \quad \text{since } n \geq 2 \\ &= cn \lg n - cn - c \lg(n-1) - cn + c + \Theta(n) \\ &= cn \lg n - (2cn + c \lg(n-1) - c) + \Theta(n) \\ &\geq cn \lg n. \end{aligned}$$

since we can pick the constant c small enough so that the $\Theta(n)$ term dominates the quantity $2cn + c \lg(n-1) - c$. Thus, the best-case running time of quicksort is $\Omega(n \lg n)$.

We now show how to find the minimum value of $f(q) = q \lg q + (n-q-1) \lg(n-q-1)$ in the range $1 \leq q \leq n-1$ (see the plot on the last page). We need to find the value of q for which the derivative of f with respect to q is 0. We rewrite this function as

$$f(q) = \frac{q \ln q + (n-q-1) \ln(n-q-1)}{\ln 2},$$

and so,

$$\begin{aligned} f'(q) &= \frac{d}{dq} \left(\frac{q \ln q + (n-q-1) \ln(n-q-1)}{\ln 2} \right) \\ &= \frac{\ln q + 1 - \ln(n-q-1) - 1}{\ln 2} \\ &= \frac{\ln q - \ln(n-q-1)}{\ln 2}. \end{aligned}$$

The derivative $f'(q)$ is 0 when $q = n - q - 1$, or when $q = (n-1)/2$. To verify that $q = (n-1)/2$ is indeed a minimum (not a maximum or an inflection point), we need to check that the second derivative of f is positive at $q = (n-1)/2$:

$$\begin{aligned} f''(q) &= \frac{d}{dq} \left(\frac{\ln q - \ln(n-q-1)}{\ln 2} \right) \\ &= \frac{1}{\ln 2} \left(\frac{1}{q} + \frac{1}{n-q-1} \right) \\ f''\left(\frac{n-1}{2}\right) &= \frac{1}{\ln 2} \left(\frac{2}{n-1} + \frac{2}{n-1} \right) \\ &= \frac{1}{\ln 2} \frac{4}{n-1} \\ &> 0 \quad (\text{since } n \geq 2). \end{aligned}$$

Problem 4. 7.4-5 p.159 quicksort with insertion sort

When quicksort is called on a subarray with fewer than k elements, let it simply return without sorting the subarray. After the top-level call to quicksort returns, run insertion sort on the entire array to finish the sorting process. We will show that this algorithm runs in $O(nk + n \lg(n/k))$ expected time.

Insertion sort's cost is $O(nk)$ in the worst case After quicksort is done the array is partitioned into sections of size from 1..k. These sections are unsorted but if a section A lies before a section B, then all numbers in A are \leq all numbers in B. Therefore when insertion sort is run, all insertions happen within the

same sections. Since the sections have at most k elements, the number of comparisons per elements is $O(nk)$ in the worst case. The analysis of the average case is trickier.

Quick sort's cost is $O(n \lg(n/k))$ in the average case

Solution 1: Use the recurrence developed in class for the average case number of comparisons of Quick sort:

$$T(n) = n - 1 + \frac{1}{n} \sum_{i=1}^n (T(j-1) + T(n-j)) = n - 1 + \frac{2}{n} \sum_{j=1}^{n-1} T(j).$$

Previously, $T(1) = T(0) = 0$. To compute the contribution of Quicksort when early stopping is used, set $T(j) = 0$, for $j = 0..k$. Follow the same steps as in class:

$$T(n) = (n+1)v(n), \text{ where } v(n) = \sum_{i=k}^n \frac{2}{n} + \Theta(1) = n \ln \frac{n}{k} + \Theta(1).$$

Therefore, $T(n) = 2n \ln \frac{n}{k} + \Theta(n)$.

Solution 2: To show that quicksort's contribution to the expected running-time is $O(n \lg n/k)$ we modify the argument given in section 7.4.2 of Cormen that proves the expected running time of quicksort on an n element list is $O(n \lg n)$.

On page 158, the book changes variables on the inner sum letting $k = j - i$. To avoid confusion, we will use a variable other than k for the index of the sum because we don't want k to have multiple meanings. Instead, let $m = j - i$.

The probability that two elements z_i and z_j are compared, is the probability that z_i or z_j is the first element chosen as a pivot from the set $Z_{i,j} = \{z_i, z_{i+1}, \dots, z_{j-1}, z_j\}$. If $j - i + 1 \leq k$ then $P(z_i \text{ and } z_j \text{ are compared}) = 0$. So (by modifying the argument given on page 158) we have:

$$\begin{aligned} E(x) &\leq \sum_{i=1}^{n-1} \sum_{m=k}^n 2/m \\ &= \sum_{i=1}^{n-1} 2(\ln n - \ln k) \quad (\text{by approx of harmonic num}) \\ &= O(n \lg \frac{n}{k}) \end{aligned}$$

Total average case cost By summing the contribution of both algs we get $O(n \lg(n/2) + nk)$ on the average.

How to choose k : In practice, experimentally find at what k insertion sort starts to beat quick sort on the average. For this purpose, create random permutations of $1..k$ and compare the *running time of your the algs on your particular computer*. Compute the variance of your average time estimates as well. Most

likely the cross over value for k will be small enough so that you can average over all $k!$ permutations.

The most sensible theoretical choice would be to compute the crossover point of the *average case number of comparisons* of quick sort and insertion sort:

$$2k \ln k = \frac{k^2}{4}.$$

However, the above estimates are too rough (the cross over is around $k = 1$).

More exact comparison of the recurrences we did in class for both algs:

$$(k+1) \left(\sum_{i=1}^k \frac{4}{i+1} - \sum_{i=1}^n \frac{2}{i} \right) = \frac{k(k-1)}{4} + k - 1 - \sum_{j=2}^n \frac{1}{j}.$$

Now the cross over is around $k = 3$.

Better to compute the average number of comparisons of both algorithms on all permutations, for $k = 2, \dots, 10$. The crossover might be slightly larger than $k = 3$.

Problem 5. 7-4 p.162 quicksort with tail recursion

a. QUICKSORT' does exactly what QUICKSORT does; hence it sorts correctly. QUICKSORT and QUICKSORT' do the same partitioning, and then each calls itself with arguments $A, p, q - 1$. QUICKSORT then calls itself again, with the arguments $A, q + 1, r$. QUICKSORT' instead sets $p \leftarrow q + 1$ and performs another iteration of its **while** loop. This executes the same operations as calling itself with $A, q + 1, r$, because in both cases, the first and third arguments (A and r) have the same values as before, and p has the old value of $q + 1$.

b. The stack depth of QUICKSORT' will be $\Theta(n)$ on an n -element input array if there are $\Theta(n)$ recursive calls to QUICKSORT'. This happens if every call to PARTITION(A, p, r) returns $q = r$. The sequence of recursive calls in this scenario is

QUICKSORT'($A, 1, n$),
 QUICKSORT'($A, 1, n - 1$),
 QUICKSORT'($A, 1, n - 2$),
 ⋮
 QUICKSORT'($A, 1, 1$)

Any array that is already sorted in increasing order will cause QUICKSORT' to behave this way.

c. The problem demonstrated by the scenario in part (b) is that each invocation of QUICKSORT' calls QUICKSORT' again with almost the same range. To avoid such behavior, we must change QUICKSORT' so that the recursive call is on a smaller interval of the array. The following variation of QUICKSORT' checks which of the two subarrays returned from PARTITION is smaller and recurses on the smaller subarray, which is at most half the size of the current array. Since the array size is reduced by at least half on each recursive call,

the number of recursive calls, and hence the stack depth, is $\Theta(n)$ in the worst case. Note that this method works no matter how partitioning is performed (as long as the PARTITION procedure has the same functionality as the procedure given in Section 7.1).

QUICKSORT''(A, p, r)

1. **while** $p < r$
2. **do** (Partition and sort the small subarray first)
3. $q \leftarrow \text{PARTITION}(A, p, r)$
4. **if** $q - p < r - q$
5. **then** QUICKSORT''($A, p, q - 1$)
6. $p \leftarrow q + 1$
7. **else** QUICKSORT''($A, q + 1, r$)
8. $r \leftarrow q - 1$

The expected running time is not affected, because exactly the same work is done as before: the same partitions are produced, and the same subarrays are sorted.

Extra Credit. 6.5-8 p.142 Merge k sorted lists with a total of n elements.

MERGE-SORTED

Build heap with first elements of the k lists. (cost $O(k)$ in total)

All elements in the heap have a pointer to successor in the list they came from (nil if there is no successor).

while heap not empty

Extract min from heap and append to output

(cost $O(\lg k)$ per element)

If successor of output element \neq nil, insert successor into heap (cost $O(\lg k)$ per element)

Since we have n elements in all the lists, the overall time is $O(n \log k)$.

```
> f:=(q*log(q)+(n-q-1)*log(n-q-1))/log(2);
```

$$f := \frac{q \ln(q) + (n - q - 1) \ln(n - q - 1)}{\ln(2)}$$

```
> fp:=diff(f,q);sol:=solve(fp,q);
```

$$fp := \frac{\ln(q) - \ln(n - q - 1)}{\ln(2)}$$

$$sol := \frac{1}{2}n - \frac{1}{2}$$

```
> fpp:=diff(fp,q);simplify(subs(q=sol,fpp));
```

$$fpp := \frac{\frac{1}{q} + \frac{1}{n - q - 1}}{\ln(2)}$$

$$\frac{4}{(n - 1) \ln(2)}$$

```
> plot(subs(n=100,f),q=1..100);
```

```
>
```

