Illinois Institute of Technology
Department of Computer Science

Solutions to First Examination

CS 430 Introduction to Algorithms
Spring, 2016

Monday, February 15, 2016
10am–11:15am & 11:25am–12:40pm, 111 Life Sciences

Exam Statistics

123 students took the exam. The range of scores was 17–90, with a mean of 51.93, a median of 52, and a standard deviation of 17.11. Very roughly speaking, if I had to assign final grades on the basis of this exam only, above 70 would be an A (19), 50–69 a B (48), 35–49 a C (37), 21–34 a D (13), below 21 an E (6). Every student should have been able to get full credit, or nearly so, on the first and last problems, plus a few points on the other three problems; that is, no score should have been below 40.

Problem Solutions

1. For a decrease by half:

   (a) n^2 becomes $(n/2)^2 = n^2/4$ so the algorithm runs 4 times as fast

   (b) n^3 becomes $(n/2)^3 = n^3/8$ so the algorithm runs 8 times as fast

   (c) $100n$ becomes $100 \cdot (n/2) = 50n$ so the algorithm runs twice as fast

   (d) $n\lceil \lg n \rceil$ becomes $(n/2)\lceil \lg(n/2) \rceil = (n\lceil \lg n \rceil)/2 - n/2$ so the algorithm runs a bit more than twice as fast as n gets large

   (e) 2^n becomes $2^{n/2} = \sqrt{2^n}$ so the algorithm runs in the *square root* of the original time

   For a decrease by 1:

   (a) n^2 becomes $(n-1)^2 = n^2 - 2n + 1$ so the algorithm saves $(2n-1)$ units of time

   (b) n^3 becomes $(n-1)^3 = n^3 - 3n^2 + 3n - 1$ so the algorithm saves $(3n^2 - 3n + 1)$ units of time

   (c) $100n$ becomes $100(n-1) = 100n - 100$ so the algorithm saves 100 units of time

   (d) $n\lceil \lg n \rceil$ becomes $(n-1)\lceil \lg(n-1) \rceil \approx n\lceil \lg n \rceil - \lceil \lg n \rceil$ so the algorithm saves about $\lg n$ units of time

   (e) 2^n becomes $2^{n-1} = 2^n/2$ so the algorithm runs twice as fast

2. Following the hint, we modify MERGE-SORT (page 34 of CLRS) sort to count the number of inversions (pairs (i, j) for which $i < j$ and $A[i] > A[j]$ in $\Theta(n \log n)$ time.

   To start, define a *merge-inversion* as a situation within the execution of merge sort in which the MERGE procedure (page 31 of CLRS), after copying $A[p..q]$ to L and $A[q+1..r]$ to R, has values x in L and y in R such that $x > y$. Consider an inversion (i, j), and let $x = A[i]$ and $y = A[j]$, so that $i < j$ and $x > y$. We claim that if we were to run merge sort, there would be exactly one merge inversion involving x and y. To see why, observe that the only way in which array elements change their positions is within the MERGE procedure. Moreover, since MERGE keeps elements within L in the same relative order to each other, and correspondingly for R, the only way in which two elements can change their ordering relative to each other is for the greater one to appear in L and the lesser one to appear in R. Thus, there is at least one merge-inversion involving x and y. To see that there is exactly one such merge-inversion, observe that after any call of MERGE that involves both x and y, they are in the same sorted sub array and will therefore both appear in L or both appear in R in any given call thereafter. Thus, we have proven the claim. We have shown that every inversion implies one merge-inversion. In fact, the correspondence between inversions and merge-inversions is one-to-one. Suppose we have a merge-inversion involving values x and y, where x originally was $A[i]$ and y was originally $A[j]$. Since we have a merge-inversion, $x > y$. And since x is in L and y is in R, x must be within a sub array preceding the sub array containing y. Therefore x started out in a position i preceding y's original position j, and so (i, j) is an inversion.

   Having shown a one-to-one correspondence between inversions and merge inversions, it suffices for us to count merge-inversions. Consider a merge-inversion involving y in R. Let z be the smallest value in L that is greater than y. At some point during the merging process, z and y will be the "exposed" values in L and R, that is, we will have $z = L[i]$ and $y = R[j]$ in line 13 of MERGE. At that time, there will be merge-inversions involving y and $L[i], L[i+1], L[i+2], \cdots, L[n_1]$, and these $n_1 - i + 1$ merge-inversions will be the only ones involving y. Therefore, we need to detect the first time that z and y become exposed during the MERGE procedure and add the value of $n_1 - i + 1$ at that time to our total count of merge-inversions.

3. (a) Compare each nut with each bolt. Since there are n nuts and n bolts, that will take $\Theta(n^2)$ comparisons in the worst case.

   (b) Assume that the nuts are labeled with numbers $1, 2, \cdots, n$ and so are the bolts. The numbers are arbitrary and do not correspond to the volumes of the items, but are just used to refer to the nuts or bolts in the algorithm description. Moreover, the output of the algorithm will consist of n distinct pairs (i, j), where the nut i and the bolt j have the same volume. The procedure MATCH-NB takes as input two sets representing nuts and bolts to be matched: $N \subseteq 1, \cdots, n$, representing nuts, and $B \subseteq 1, \cdots, n$, representing bolts. We will call the procedure only with inputs that can be matched; one necessary condition is that $|N| = |B|$.

Algorithm 1 MATCH-NB(N, B)

```
1: if |N| == 0 then
2:    return
3: end if
4: if |N| == 1 then
5:    let N = n and B = b output "(n, b)"
6:    return
7: end if
8: n = a randomly chosen nut in N
9: compare n to every bolt of B
10: B_< = the set of bolts in B that are smaller than n
11: B_> = the set of bolts in B that are larger than n
12: b = the one bolt in B with the same size as n
13: compare b to every nut of N − n
14: N_< = the set of nuts in N that are smaller than b
15: N_> = the set of nuts in N that are larger than b
16: output "(n, b)".
17: MATCH-NB(N_<, B_<)
18: MATCH-NB(N_>,B_>)
```

4. Below is code that implements the insert operation. It is very much like the code for a binary max-heap: we insert the key at the next available child spot in the heap (the next element in the array) and we "bubble" the key upward. We call one of two different bubble up routines depending on the level in the heap that we perform the initial insert, and depending on the inserted keys value relative to the keys of its immediate ancestors, that is, its parent and grandparent.

   The supporting BubbleUp code is below. In it, the value moves up by two levels in the heap with each exchange, rather than just one level.

Algorithm 2 InsertMinMax(k, H)

```
1: size(H) = size(H) + 1
2: i = size(H)
3: H[i] = k
4: if i is on a max level then
5:    if Parent(i) > 0 and H[Parent(i)] > H[i] then
6:       exchange H[i] with H[Parent(i)]
7:       BubbleUpMin(Parent(i), H)
8:    else
9:       BubbleUpMax(i, H)
10:   end if
11: else
12:   {i is on a min level}
13:   if Parent(i) > 0 and H[Parent(i)] < H[i] then
14:      exchange H[i] with H[Parent(i)]
15:      BubbleUpMax(Parent(i), H)
16:   else
17:      BubbleUpMin(i, H)
18:   end if
19: end if
```

Algorithm 3 BubbleUpMin(i, H)

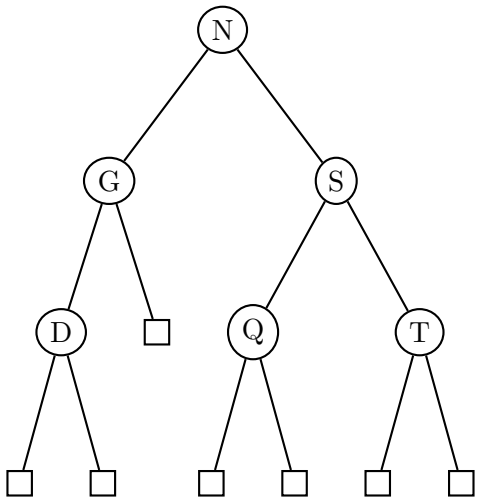
```
1: if Parent(Parent(i)) > 0 and H[Parent(Parent(i))] > H[i] then
2:    exchange H[i] with H[Parent(Parent(i))]
3:    BubbleUpMin(Parent(Parent(i)), H)
4: end if
```

Algorithm 4 BubbleUpMax(i, H)

```
1: if Parent(Parent(i)) > 0 and H[Parent(Parent(i))] < H[i] then
2:    exchange H[i] with H[Parent(Parent(i))]
3:    BubbleUpMax(Parent(Parent(i)), H)
4: end if
```

5. (a) The external path length is $3 + 3 + 2 + 2 + 4 + 4 + 3 = 21$.

   (b) Yes; we saw (in class on January 20) that the binary tree with the least external path length has all of its leaves on level l or level $l + 1$, for some l. Thus we can reorganize the tree (rotating counter clockwise at node Q) to



   which has external path length $3 + 3 + 2 + 3 + 3 + 3 + 3 = 20$.

   (c) No. The root must be black. Then G must be black (if it were red, D would have to be black and the leaf at the right of G would have black-depth 1, but the leaves below D would have black depth 2) and D must be red—this gives the leaves in the left subtree of N black depth 2. For the leaf to the left of Q to have black depth 2, Q must be black. But at least one of T and S must be black (because we cannot have two red nodes in a row), meaning that the leaves below S would have black depth 3.