CPSC 320 Sample Midterm 2
November 2013

[9] 1. Sorting and Order Statistics

[3] a. Why is *RandomizedQuicksort* (Quicksort using a random pivot) better than the regular (non-randomized) version of Quicksort?

Solution : Because it performs well on average for every possible input, unlike Quicksort which performs well (all the time) for some inputs, and badly (all the time) for other inputs such as already sorted arrays.

[6] b. In assignment 6, you wrote an algorithm to find the i^{th} order statistics of a set stored in a binary search tree. Recall that we used an additional piece of data stored in each node N of the tree: the size of the subtree rooted at N. Describe an algorithm that determines, given such a binary search tree T and a key k, the *rank* of k in the set stored in T (that is, if k is the 17^{th} smallest element of the set, then your algorithm should return 17).

Solution :

```
Algorithm findRank(N, k)
  //
  // Case 1: the key is not in the tree.
  //
  if N is null then
    return -1

  //
  // Size of left subtree.
  //
  if N has a left child then
    leftsize ← leftChild[N].size
  else
    leftsize ← 0

  //
  // Case 2: it's in the current node.
  //
  if k = key[N] then
    return leftsize + 1

  //
  // Case 3: it's in the left subtree.
  //
  if k < key[N] then
    return findRank(leftChild[N], k)
```

```
//
// Case 4: it's in the right subtree.
//
return leftsize + 1 + findRank(rightChild[N], k)
```

[10] 2. Amortized Analysis

[3] a. Can amortized analysis help us derive a better upper-bound on the worst-case running time of a single operation on a data structure? Why or why not?

Solution : No: it is only used for sequences of operations on the data structure, because it works by "borrowing" time from one operation to pay for others.

[3] b. You are using the potential method to obtain a tight upper bound on the running time of a sequence of operations on a data structure. The *brziptle* operation on this (bizarre) data structure executes a constant number of constant time steps, as well as one loop. This loop iterates j times, where the value of j varies from one *brziptle* call to the next, and might be as large as $n-1$ (where n is the number of elements in the data structure). Each loop iteration runs in $\Theta(1)$ time.

Explain what you would need to do in order to prove that the amortized cost of the *brziptle* operation is in $\Theta(1)$.

Solution : You would need to show that the *brziptle* operation reduces the potential of the data structure by $j \pm \Theta(1)$.

[4] c. Consider a data structure that supports the following three operations:

- `insert(x)`, that runs in $\Theta(1)$ time.
- `search(x)`, that runs in $O(j)$ time where j is the number of elements that it will visit before finding x (in the worst case, j could be the number of elements of the data structure).
- `delete(x)`, that first calls `search(x)` and then performs a $\Theta(1)$ time deletion.

Assume furthermore that elements "wear out": that is, the fourth time that an elements is visited during a `search` operation, it is deleted (in $\Theta(1)$ time).

Suggest a potential function that could be used to prove that the amortized cost of every operation on this data structure is in $\Theta(1)$ (you do **not** need to compute the amortized costs). Justify your answer briefly.

Solution : We would use $\Phi(D_i) = \sum_{x \in D_i} \phi(x)$ where $\phi(x) = 4 - f(x)$, where $f(x)$ is the number of times that element x has been visited. This way, the potential in the elements visited by the `search` operation will pay for the visit, and the amortized cost of `search` will be 0.

[4] 3. Explain the best way to find a tight upper bound on the solution of the recurrence relation

$$T(n) = \begin{cases} 4T(n/2+3) + 5n^2 \log n & \text{if } n \geq 8 \\ \Theta(1) & \text{if } n < 7 \end{cases}$$

and state what you believe the upper bound to be (you do not need to prove your upper bound).

Solution : We would first use the Master theorem to obtain a bound (ignoring the $+3$ term) and then use guess and test with this bound to prove it formally.

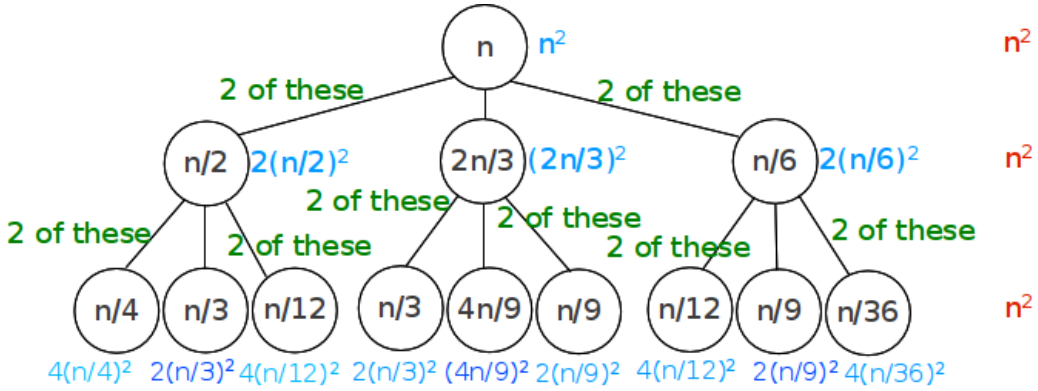
Now, if we ignore the $+3$ term, we have $a = 4$, $b = 2$, and $n^{\log_b a} = n^{\log_2 4} = n^2$. Hence $f(n) \in \Theta(n^{\log_b a} \log n)$ which means that we are in case 2, and so our guess would be $T(n) \in \Theta(n^2 \log^2 n)$.

[8] 4. Prove an upper bound on the function $T(n)$ defined by

$$T(n) = \begin{cases} 2T(n/2) + T(2n/3) + 2T(n/6) + n^2 & \text{if } n \geq 6 \\ 1 & \text{if } n \leq 5 \end{cases}$$

Your grade will depend on the quality of the bound you provide (that is, showing that $T(n) \in O(100^n)$, while true, will not give you many marks).

Solution : Here are the first three levels of the recursion tree (some of the repeated nodes have been removed, and replaced by an indication that there are "2 of these" since the tree would not have fit on the page otherwise).



As we can see, the children of a node do the same amount of work, together, as their parent. Hence the work done on every level of the tree is exactly n^2, at least up to the level containing the leaf closest to the root (after which the amount of work done starts decreasing). The tree contains $\log_{3/2} n$ levels, since the path along which the size of the subproblems decreases slowest is that where the size is multiplied by $2/3$ when we go from one level to the next. Hence the total amount of work done is in $O(n^2 \log n)$ (recall that the base of the

$\log$ does not matter when we use asymptotic notations, because $\log_x n$ is within a constant factor of $\log_y n$).

For the lower bound (which you were not asked to provide), notice that the first leaf occurs at level $\log_6 n$, and that every level up to that one does exactly n^2 work, and so the total amount of work done is in $\Omega(n^2 \log n)$.

[9] 5. Design a divide-and-conquer algorithm that takes as input an array of integers, and returns both the minimum and the maximum elements of the array. Your algorithm should make at most $3n/2 + c$ comparisons where c is some small (additive) constant.

Solution :

```
Algorithm MinAndMax(A, first, last)
if (first = last) then
    return (A[first], A[first])

mid ← (first + last)/2
pair1 ← MinAndMax(A, first, mid)
pair2 ← MinAndMax(A, mid + 1, last)

return (min(pair1[0], pair2[0]), max(pair1[1], pair2[1]))
```