Homework 4

Matt Forbes

November 16, 2010

Problem 1

a) A linked list provides a very good representation of the mechanics of the game. The list is the 'circle' that the kids are lined up in, and simulating the game is as easy as iterating through the list and removing every w^{th} kid.

To start with, we would need to build the list. Each kid gets his own node in the list, and they are inserted in order (where order is determined by the circle, original caller is 1, his right is 2, etc.). This would take $O(p)$ time as there are p kids.

Simulating the selection requires $p-1$ deletions from the list. One for each kid that is eliminated. Starting from the head of the list, follow $w-1$ links in the list. Then delete the next link. Now, follow $w-1$ links again, and then delete the next. Whenever you reach the end of the list, wrap around to the first element (counting that as one visit). Continue doing this until there is only one link left in the list. That last element remaining is the kid who is "it." In this step, we followed $w-1$ pointers exactly $p-1$ times, which is $O(pw)$.

Adding the running times of both the creation of the list and elimination phase gives $O(p+pw)$ which is more simply just $O(pw)$.

b) Using OS_Tree developed in class to implement emmm_select:

Pseduo Code

```
(* emmm_select performs eeny-meeny-miny-mo selection on the
array of p kids (kid_arr), using a song with w words *)

def emmm_select(kids, p, w):
    T is a new OS_Tree of size p
    for i in 1...p:
        insert(T, kids[i])

    caller = 1
    size = p
```

```
while size > 1:
    (* kid going out is w-1 kids away from the caller, with wrapping *)
    out = (caller + w-1) % (total + 1)
    size = size - 1
    kid = OS_Select(T, out)
    remove(T, kid)

    (* next caller is kid who takes out's position in circle *)
    caller = out

(* return rank 1 (only) kid left in tree *)
return OS_Select(T, 1)
```

Justification

emmm_select relies on the OS_Tree to perform the rank lookups in $O(\log p)$ time to achieve its running time of $O(p \log p)$. Similarly to the linked-list version, the first step is to build the data structure. In this case, it performs p inserts that require $O(\log p)$ time each, resulting in a $O(p \log p)$ step.

Next is the actual elimination phase. In general, the kid that is going to be eliminated is $w - 1$ ranks above the current caller, where rank is the order around the circle. So we keep track of the caller at each iteration, and simple eliminate a kid each time until there is only one remaining in the tree. After a kid is eliminated the next caller is going to be the element with rank now equal to what we just removed (just the next kid in line). So the process repeats, always making sure not to remove an element with rank that is out of range. To avoid this, the next caller's rank is always 'modded' with 1+ the remaining kids. This is repeated $p - 1$ times so that there is only kid left in the circle, who is now it. The most expensive part of this step is finding and removing the correct element. Searching by rank is $O(\log p)$ given by OS_Tree, and remove is also $O(\log p)$ given by the balanced binary tree. Other operations are in constant time. There are $p - 1$ iterations, thus $p - 1$ searches/removes, so the total running time of this part is $O(p \log p)$.

Both steps of this algorithm run in $O(p \log p)$ time, so the whole algorithm must be bounded by $p \log p$.

Problem 2

Insert Pseudocode

```
(* assumed definitions: *)
(* add_to_end - create p-heap node at end of H with given items and return it *)
(* get_parent - return parent of given node *)
(* get_elem - get ith element out of a node *)
(* swap(node,i,node2,j)  - swap ith element of node with jth element of node2 *)


(* insert elems (array length p) in to p-heap H *)
def insert(H, p, elems):
    node = add_to_end(H, elems)
    for i in 1...p:
        bubble_up(node, i, p)

def bubble_up(node, i, p):
    parent = get_parent(node)
    elem = get_elem(node, i)
    best = infinity
    best_ix = nil
    for j in 1...p:
        parent_elem = get_elem(parent, j)
        if elem > parent_elem and parent_elem < best:
            best = parent_elem
            best_ix = j
    if best_ix is not nil:
        swap(node, i, parent, best_ix)
        bubble_up(parent, best_ix, p)
```

Insert works by creating a new node in the very last position of the heap that holds all p values to be inserted. Each of these values is then 'bubbled-up' to the top of the heap just like a normal heap. As an element is bubbling up, it should only move up if it is greater than at least one of the values in its parent. The value that it replaces must be the smallest value in the parent that is smaller than this element. Because it must check all p value in its parent, each bubble step is $O(p)$. In the worst case, each inserted value would need to be bubbled all the way to the root of the tree, which would mean at most $\log_2 \frac{n}{p}$ bubble ups. So, the worst case running time for a bubble_up is $O(p \log_2 \frac{n}{p})$.
A call to insert means p full bubble ups. So the running time of insert must be $(p^2 \log_2 \frac{n}{p})$.

Extract Max Pseudocode

```
(* assumed definitions: *)
(* get_elem - get ith element out of a node *)
(* swap(node,i,node2,j)  - swap ith element of node with jth element of node2 *)
(* left/right - get left or right subtree of a node *)
(* detach_root - remove root node from heap and return it *)
(* attach_root - set root of the tree to given node *)
(* detach_last - remove last node from heap and return it *)
(* items - return items on a node *)

(* remove p greatest elements in heap *)
def extract_max(H, p):
    top = detach_root(H)
    last = detach_last(H)
    attach_root(H, last)
    for i in 1...p:
        bubble_down(last, i)
    return items(top)

def bubble_down(node, i, p):
    best = get_element(node, i)
    best_ix = nil
    best_node = nil
    for j in 1...p:
        check = get_element(left(node), j)
        if check > best:
            best = check
            best_ix = j
            best_node = left(node)
        check = get_element(right(node), j)
        if check > best:
            best = check
            best_ix = j
            best_node = right(node)
    if best_ix is not nil:
        swap(node, i, best_node, best_ix)
        bubble_down(best_node, best_ix, p)
```

Extract_max is similar to insert but it has to bubble down the tree rather than up it. It starts off by removing the root element so its elements can be returned, and then moves the very last node up and attaches it to the root. That preserves the 'complete' requirements of the tree, but the elements on this new root are definitely not the p largest in the tree. So we bubble each p value down the tree to a spot where it actually fits.

Bubble_down finds the maximum value out of both it's children's values first. If this maximum value is greater than the value we are trying to bubble down, it swaps the two. Now this node satisfies the heap constraint because we put the largest value in the parent. If we swapped, there is no guarantee that we didn't screw up the node that we swapped in to, so we need to try bubbling the value down again. So we just recur in to the child node at the swapped index. It can only recurse a maximum of $\log_2 \frac{n}{p}$ times (the height of the tree). Each bubble_down call takes $O(p)$ time because we need to check all p values of both the children nodes to find the maximum. So, the running time of bubble_down is $O(p \log_2 \frac{n}{p})$.
The algorithm bubbles down on all p values that are moved to the root, so we are looking at a running time of $O(p^2 \log_2 \frac{n}{p})$.
Both of these algorithms have the same running time if $O(p^2 \log_2 \frac{n}{p})$. But p is generally going to be much smaller than n, so we could probably just call it a constant. By assuming p is constant compared to n, the new running time would be $O(\log_2 n)$.

Problem 3

The goal of this algorithm is to sort all m strings which have a total of n characters in $O(n)$ time. To accomplish this, my plan is use a sort that has aspects of both radix sort and merge sort.

If we do a radix sort using the first character as the key, our array will look something like this:

$$[\text{“}a...\text{”}, \text{“}a...\text{”}, ..., \text{“}b...\text{”}, \text{“}b...\text{”}, ..., \text{“}z...\text{”}]$$

So the array will now be grouped in to the a's and then the b's and then the c's, etc. Each group is in order relative to each other. So in the final sorted order, all of the words that start with 'a' will still be in those first slots of the array.

Now if while we were doing the radix sort, we found the index ranges of each 'string-group' at the same time, we wouldn't be adding any extra running time to this step. A very convenient property of radix sort is that right before we build up our final sorted array B, we have the auxillary array C that stores the last index of each occurrence in to the final array. After doing the rest of the sort, that same array C now stores the first index of each occurrence. How does this help? Well those values are the ranges of what we were trying to keep track of!

In C's first state, C['a'] is the index of the last occurrence of a string starting with 'a' in the final sorted array. Likewise for all other characters. After the sort is finished, C['a'] is now the first occurrence of a string starting with 'a' in the final sorted array. We just easily found the indices of all strings starting with 'a' in our final sorted array.

We can now recursively call our sorting algorithm on each subset of the array that we just found. If we sort all of the a's only with respect to each other, all the b's with respect to each other, etc. we will have a completely sorted array.

Each level of recursion will sort on the next character position unless a group has only one member, because an array of size one is sorted trivially.

Each time a group is formed on a nul character that was used for padding, we don't need to sort that group because it is in it's final position in the array. So on the next level of recursion, we don't have to keep comparing against a nul character which would end up making the running time worse than $O(n)$.

Some rough pseudocode should solidify the algorithm.

```
(* g_radix_sort is the basic radix sort that returns the
string-groups described above. The format of the string
groups would be an array whose elements are the indices of the
first and last occurrence of a string starting with that
character position. The last group (27th index) represents
the group starting with the nul character *)

(* sort arr from index first to last using radix style sort on key *)
def str_sort(arr, first, last, key):
    groups = g_radix_sort(arr, first, last, key)
    for i in 1...26:
        front = groups[i][1]
        back = groups[i][2]
        if (back - front) > 1:
            str_sort(arr, front, back, key+1)
```

There are a couple of cases that could be considered for the worst case analysis. One of which is when we have one really long word, and a bunch of really small ones, say just one character each. Another case could be if all the strings are equally long. In this scenario, we would sort on each index of the strings all the way to the last before we make any progress.

Well, having one very large string in the array is actually a best case for this algorithm. After sorting on the first key, all strings are in their final order except for the very long string and those that start with the same letter as it (the string-group). Now we only need to sort the string-group that the long string is a part of. Sorting this group is also very easy, because if all strings besides the larger one only have one character, they are padded with the nul character for this radix sort. So when this is finished, the long string will be at the end of the group and all the others will be before it, which is sorted order. So in total, this case took $O(m)$ for the first radix sort, and $O(m)$ for the second, so $O(m)$ in total. $m \leq n$ so $O(m) \leq O(n)$.

Now the worst case is if all the strings are equally long. This algorithm is of the divide and conquer flavor, but when the strings are all the same up until the last character, we are not splitting up the problem at all. Well, the length of each string l, must be $\frac{n}{m}$ because they are of equal length. The algorithm will run a radix sort on $l-1$ keys before changing any ordering. The final step is where the real work is done, and that is just one more radix sort. Each sort runs in $O(m)$ time, and we need to sort l times. Well we said that l is just $\frac{n}{m}$, so the total running time is $O(n)$ which was our target.