Algorithms and Data: CS 4800 Fall '12
December 14, 2012

CS 4800 Fall '12 Sample Final

Instructions: You must complete this exam in the time provided. You may use your textbooks and any printed notes, homeworks, or prior exams you brought with you. You may not use any electronic devices during the exam period. You may not communicate with other students during the exam period.

The exam has 5 questions worth 20 each. Your exam score will be the total of your 3 best individual question scores; the highest possible exam score is therefore 60.

Problem 1: Quicksort

The quicksort algorithm runs in $O(n \lg n)$ best-case time and $O(n^2)$ worst-case time. The difference in performance depends on the choice of *pivot* in each step: the element of the input used to partition the remaining elements. Reliably choosing a good pivot can improve quicksort's running time and avoid $O(n^2)$ worst-case time.

1. In the worst case, the recursion tree for quicksort is $O(n)$ deep. Assuming we could reliably choose a good pivot, we want to determine how good our choice must be to improve the recursion depth. Consider the following lower bounds for the size of the smaller side of a partition. In each case, report the worst-case recursion depth of the resulting quicksort. For instance, in normal quicksort the smaller side may have 0 elements, and the worst-case recursion depth is $O(n)$. Optimal quicksort, if we somehow always pick the median element as our pivot, guarantees the smaller side has $\lfloor n/2 \rfloor$ elements and has the worst-case recursion depth of $O(\lg n)$.
   (a) 10 elements
   (b) $\lg n$ elements
   (c) $\sqrt{n}$ elements
   (d) $\lfloor n/3 \rfloor$ elements
2. Let us assume we can somehow pick *the median element* as our pivot. We want to determine how fast our pivot selection must be to achieve $O(n \lg n)$ worst-case time for quicksort. Consider the following running times for our pivot-selection algorithm. In each case, report the worst-case running time for quicksort using the given speed for selection.
   (a) $O(n^2)$
   (b) $O(n\sqrt{n})$
   (c) $O(n \lg n)$
   (d) $O(n)$
   (e) $O(\sqrt{n})$

Solution:

1. (a) $O(n)$
   (b) $O(n/\lg n)$
   (c) $O(\sqrt{n})$
   (d) $O(\lg n)$
2. (a) $O(n^2)$
   (b) $O(n\sqrt{n})$
   (c) $O(n(\lg n)^2)$
   (d) $O(n \lg n)$
   (e) $O(n \lg n)$

Problem 2: Minimum Spanning Trees

For both of the following subproblems, we will describe a weighted, undirected graph by giving only the weights of its edges. The graph will therefore have all of the vertices and edges for which weights are given, and no others.

1. Report all possible minimum spanning trees for the graph with the following weights: $w(a,b) = 1$, $w(a,c) = 2$, $w(a,d) = 2$, $w(a,e) = 1$, $w(b,c) = 1$, $w(b,d) = 2$, $w(b,e) = 2$, $w(c,d) = 1$, $w(c,e) = 2$, $w(d,e) = 1$.
2. Consider the graph with the following weights: $w(a,b) = 1$, $w(a,c) = 2$, $w(a,d) = 6$, $w(a,e) = 9$, $w(b,c) = 3$, $w(b,d) = 4$, $w(b,e) = 10$, $w(c,d) = 5$, $w(c,e) = 8$, $w(d,e) = 7$.
   (a) What minimum spanning tree would Kruskal's algorithm produce? Write the edges in the order that the algorithm would add them to its result. If there are multiple possible trees or orders, any one that the algorithm might produce is acceptable.
   (b) What minimum spanning tree would Prim's algorithm produce, starting at vertex b? Write the edges in the order that the algorithm would add them to its result. If there are multiple possible trees or orders, any one that the algorithm might produce is acceptable.

Solution:

1.
   - $(a,b),(b,c),(c,d),(d,e)$
   - $(b,c),(c,d),(d,e),(e,a)$
   - $(c,d),(d,e),(e,a),(a,b)$
   - $(d,e),(e,a),(a,b),(b,c)$
   - $(e,a),(a,b),(b,c),(c,d)$
2. (a) $(a,b),(a,c),(b,d),(d,e)$ is the only solution.
   (b) $(a,b),(a,c),(b,d),(d,e)$ is the only solution.

Problem 3: Dynamic Programming

An *addition chain* for some number n is a sequence of numbers starting with 1 and ending with n, such that each number other than the initial 1 is the sum of two previously occurring numbers. For instance, the sequence $(1,2,3,6,12,24,30,31)$ is an addition chain for 31:

$$\begin{aligned} 1+1&=2\\ 1+2&=3\\ 3+3&=6\\ 6+6&=12\\ 12+12&=24\\ 24+6&=30\\ 30+1&=31 \end{aligned}$$

A common low-level implementation for calculating exponents, x^n, is to first find the shortest possible addition chain for n. Then, for each equation $a+b=c$ that forms the chain, compute $x^a \times x^b = x^c$. This naturally concludes with x^n, in far fewer than $O(n)$ multiplications.

- To compute the addition chain for some number $n > 1$, we must in theory consider a chain whose final step is $(n-k)+k=n$ for every $0<k<n$. Write a brute-force algorithm to compute the shortest addition chain for n by solving any necessary subproblem(s) for each possible final step, then choosing the best resulting solution. What is the tightest big-O bound you can find for this solution?
- Convert your brute-force solution to a dynamic-programming solution. What are the dimensions required for the table of recorded subproblems? What subproblem does each entry in the table represent? What is the running time of the resulting algorithm? Remember to account for both the size of the recursion tree and the work done at each step.

Solution:

1. $\mathsf{min-chain}(n) =$

```
        If $n = 1$, return $\mathsf{list}(1)$.
        Otherwise, let $S = (1,2,\ldots,n)$.
        For $k \in 1,\ldots,\lfloor n/2 \rfloor$:
            Let $s_1 = \mathsf{min-chain}(k)$.
            Let $s_2 = \mathsf{min-chain}(n-k)$.
            Let $s = \mathsf{append}(\mathsf{combine}(s_1, s_2), \mathsf{list}(n))$.
            If $|s| < |S|$, set $S = s$.
        Return $S$.
```

Note that we only need to consider k up to $\lfloor n/2 \rfloor$; beyond that, the pairs $(k, n-k)$ are the same ones we have already considered, reversed.

$\mathsf{combine}(s, s') =$

```
        If $s$ is empty, return $s'$.
        If $s'$ is empty, return $s$.
        If $\mathsf{first}(s) < \mathsf{first}(s')$, then return $\mathsf{cons}(\mathsf{first}(s), \mathsf{combine}(\mathsf{rest}(s), s'))$.
        If $\mathsf{first}(s) > \mathsf{first}(s')$, then return $\mathsf{cons}(\mathsf{first}(s'), \mathsf{combine}(s, \mathsf{rest}(s')))$.
        Otherwise, if $\mathsf{first}(s) = \mathsf{first}(s')$, then return $\mathsf{combine}(\mathsf{rest}(s), s')$.
```

The function $\mathsf{combine}(s,s')$ runs in time $O(|s|+|s'|)$. The function $\mathsf{min}-\mathsf{chain}(n)$ recurs up to n times on input of size up to $n-1$; it therefore runs in time at most $O(n!)$.

2. $\mathsf{min}-\mathsf{chain}(n) =$

```
    Let $A[1,\ldots,n]$ be a fresh array.
    Set $A[1] = \mathsf{list}(1)$.
    For $i \in 2,\ldots,n$:
        Let $S = (1,2,\ldots,i)$.
        For $k \in 1,\ldots,\lfloor i/2 \rfloor$:
            Let $s_1 = A[k]$.
            Let $s_2 = A[i-k]$.
            Let $s = \mathsf{append}(\mathsf{combine}(s_1,s_2), \mathsf{list}(i))$.
            If $|s| < |S|$, set $S = s$.
        Set $A[i] = S$.
    Return $A[n]$.
```

The table has n entries representing minimum-length addition chains for every number from 1 to n. The resulting algorithm considers $n/2$ possibilities at each step and, for each one, builds a result up to length n in linear time using combine. This takes at most $O(n^3)$ total time.

Problem 4: AVL Trees

We have seen operations to insert and remove single elements from AVL trees, but sometimes we need to insert or remove many elements at once, efficiently. For instance, we might want to append two AVL trees, assuming the keys in one are strictly less than the keys in the other; we might also want to remove all keys in an AVL tree between some pair of keys k_1 and k_2. The latter operation is often possible without inspecting all of the keys to be removed; by keeping track of the upper and lower bounds on keys in the current subtree, any subtree known to be between k_1 and k_2 can be skipped entirely. The same is true of any subtree entirely outside the range from k_1 to k_2. All that remains to be done is to split any subtree that spans k_1, k_2, or both, and to recombine those subtrees that must be kept in the long run.

1. Write an algorithm for $\mathsf{remove-all}(t, k_1, k_2)$ that removes all keys between k_1 and k_2, inclusive, from t.

   You may use any operation we have seen before for AVL trees. You may also use the operation $\mathsf{append}(t_1, t_2)$ that appends two AVL trees t_1 and t_2. All of the keys in t_1 must be less than or equal to all of the keys in t_2. The operation append runs in $O(\lg|t_1| + \lg|t_2|)$ time. You do not need to write append yourself. You may not assume anything about the structure of append's result, such as its height, except that it is a valid AVL tree.
2. What is the running time of $\mathsf{remove-all}$ as you have written it?

Solution:

1. $\mathsf{remove-all}(t, k_1, k_2) = \mathsf{remove-all-within}(t, -\infty, +\infty, k_1, k_2)$

```
remove-all-within($t, K_1, K_2, k_1, k_2$) =
    If $k_1 \le K_1$ and $K_2 \le k_2$, return leaf.
    If $K_2 < k_1$ or $k_2 < K_1$, return $t$.
    If $t$ = leaf, return $t$.
    Otherwise, $t = \mathsf{node}(h, k, t_1, t_2)$.
        Let $T_1 = \mathsf{remove-all-within}(t_1, K_1, k, k_1, k_2)$.
        Let $T_2 = \mathsf{remove-all-within}(t_2, k, K_2, k_1, k_2)$.
        If $k_1 \le k \le k_2$, then return $\mathsf{append}(T_1, T_2)$.
        Otherwise, return $\mathsf{append}(T_1, \mathsf{insert}(k, T_2))$.
```

2. To analyze this algorithm, the important thing to realize is that the recursion tree only has two full-height branches. The function $\mathsf{remove-all-within}$ only continues down to a leaf for key values immediately surrounding k_1 and k_2. Any subtrees outside this path are immediately returned or discarded. The function therefore recurs at most $O(\lg n)$ times. At each step, the function does $O(\lg n)$ work via append. The function therefore takes at most $O((\lg n)^2)$ time.

Problem 5: Hash Tables

Consider an implementation for hash-tables that uses open addressing with linear probing—that is, if a key hashes to bucket i, and i is already full, the hash table checks $i+1$ next, then $i+2$, and so on, wrapping around to 0 if necessary. Assume the hash function for key k and a table of size m is simply $k \mod m$. Further assume that the hash table automatically resizes to maintain a load of at most 1/2—that is, if more than 1/2 of the slots are full, the hash table gets reallocated at a larger size, and all of the entries are hashed to new buckets. Assume that the hash table uses the following fixed sizes—whenever the table needs to be resized, it picks the next higher size on the list: 3, 5, 11, 17, 37, 67, 131, 257, 521, 1031 . . .

1. Given a hash table of size 11, provide 5 keys that will all hash to different positions, such that when the table resizes to 17 slots, all 5 hash to the same position.
2. Given a hash table of size 11, provide 5 keys that all hash to the same position, but which all hash to different positions when the table resizes to 17 slots.
3. Given two subsequent hash table sizes p_1 and p_2, both of which are prime numbers, write a function to produce $\lfloor p_1/2 \rfloor$ keys that all hash to different positions in a table of size p_1 and that all hash to the same position in a table of size p_2.
4. Given two subsequent hash table sizes p_1 and p_2, both of which are prime numbers, write a function to produce $\lfloor p_1/2 \rfloor$ keys that all hash to the same position in a table of size p_1 and that all hash to different positions in a table of size p_2.

Solution:

1. 17, 34, 51, 68, 85
2. 11, 22, 33, 44, 55
3. $\{p_2 \times 1, p_2 \times 2, p_2 \times 3, \ldots, p_2 \times \lfloor p_1/2 \rfloor\}$
4. $\{p_1 \times 1, p_1 \times 2, p_1 \times 3, \ldots, p_1 \times \lfloor p_1/2 \rfloor\}$