CS 161 - Design and Analysis of Algorithms (Summer 2012) Solutions to Homework 1
Stanford University Due: Friday, July 6, 2012 at 2:15PM

Problem 1 (15 Points). Order the following functions by big Oh. You should list the functions by increasing Big Oh, showing which functions are Big Theta of each other. For each two adjacent functions in your list, please provide a justification for the ordering (i.e. show that $f_i(n) \in O(f_j(n))$, and why $f_i(n)$ is or isn't $\Theta(f_j(n))$).

$$\begin{array}{lll}
f_1(n) = 100\sqrt{n} & f_2(n) = 2\log(10n) & f_3(n) = 3^n \\
f_4(n) = 2^n & f_5(n) = n^2 & f_6(n) = 6\log n \\
f_7(n) = n^{1/10000} & f_8(n) = \log(n^{10}) & f_9(n) = n2^n \\
f_{10}(n) = (\log n)^{\log n} & f_{11}(n) = 10n & f_{12}(n) = n^{\log n} \\
f_{13}(n) = (\log n)^{10000} & f_{14}(n) = 20n\log n & f_{15}(n) = n + 8\log n
\end{array}$$

Solution:

- $f_2(n) \in \Theta(f_6(n))$. This is because $f_2(n) = 2\log(10n) = 2\log(n) + 2\log 10 \in \Theta(\log(n))$ and $f_6(n) = 6\log n \in \Theta(\log n)$.
- $f_6(n) \in \Theta(f_8(n))$. This is because $f_8(n) = \log(n^{10}) = 10\log n \in \Theta(\log(n))$.
- $f_8(n) \in o(f_{13}(n))$. We can do the limit test: $f_{13}(n)/f_8(n) = (\log(n))^{999}/10$, whose limit is ∞.
- $f_{13}(n) \in o(f_7(n))$. Any polynomial grows faster than any power of a log.
- $f_7(n) \in o(f_1(n))$. f_1 and f_7 are both polynomials, but the exponent in f_1 (which is 1/2) is greater than the exponent in f_7 (which is 1/10000).
- $f_1(n) \in o(f_{11}(n))$. Both are polynomials, but the exponent in $f_{11}(n)$ is 1, which is greater than 1/2.
- $f_{11}(n) \in \Theta(f_{15}(n))$. $f_{11}(n) = 10n \in \Theta(n)$ and $f_{15}(n) = n + 8\log n \in \Theta(n)$.
- $f_{15}(n) \in o(f_{14}(n))$. We can do the limit test:$f_{14}(n)/f_{15}(n) = 20\frac{n\log n}{n+8\log n}$. As n goes to infinity, this approaches $20\frac{n\log n}{n} = 20\log n$, whose limit is ∞.
- $f_{14}(n) \in o(f_5(n))$. Again, we use the limit test: $f_5(n)/f_{14}(n) = \frac{n^2}{20n\log n} = \frac{n}{20\log n}$. The limit is ∞.
- $f_5(n) \in o(f_{10}(n))$. $f_{10}(n) = (\log n)^{\log n} = 2^{\log n \log\log n} = n^{\log\log n}$. Thus $f_{10}(n)/f_5(n) = n^{\log\log n - 2}$. Since $\log\log n$ goes to infinity, this quantity approaches ∞.
- $f_{10}(n) \in o(f_{12}(n))$. $f_{12}(n)/f_{10}(n) = n^{\log n - \log\log n}$. For large n, the exponent goes to ∞, so the whole fraction does.
- $f_{12}(n) \in o(f_4(n))$. $f_{12}(n) = n^{\log n} = 2^{(\log n)^2}$. Thus $f_4(n)/f_{12}(n) = 2^{n-(\log n)^2}$. For large n, the exponent goes to ∞, so the whole fraction does.

- $f_4(n) \in o(f_9(n))$. $f_9(n)/f_4(n) = n$, which goes to ∞ for large n.
- $f_9(n) \in o(f_3(n))$. $f_3(n)/f_9(n) = \frac{3}{2}^n/n$, which goes to ∞ for large n.

Problem 2 (15 Points).

(a) Suppose $f(n) \notin O(1)$. Show that $f(n)^2 \notin O(f(n))$. Conclude that there is no asymptotically largest function.*[Hint: Show the contrapositive, that if $f(n)^2 \in O(f(n))$, then $f(n) \in O(1)$]*

(b) Suppose $f(n) \notin O(1)$. Show that $2^{f(n)} \notin O(f(n))$.

Solution:

(a) Assume $f(n)^2 \in O(f(n))$. Then there exists c, n_0 such that $f(n)^2 \leq cf(n)$, which means $f(n) \leq c$. But if $f(n) \leq c$ for all $n \geq n_0$, then we have that $f(n) \in O(1)$.

(b) Assume $2^{f(n)} \in O(f(n))$. Then there exists c, n_0 such that $2^{f(n)} \leq cf(n)$. Consider the quantity $\frac{2^m}{m}$. As m goes to ∞, this quantity approaches ∞. Therefore, there is some m_0 such that $\frac{2^m}{m} > c$ for all $m \geq m_0$. Since $\frac{2^{f(n)}}{f(n)} \leq c$, it must be that $f(n) < m_0$ for all $n \geq n_0$. Thus, $f(n) \in O(1)$.

Problem 3 (5 Points). Show how to use an array to implement a queue with all operations being (amortized) constant time.

Solution: In addition to keeping track of the number of elements in the array, we will also keep track of the starting index. When we add an element to the queue, we add it to the index `start_index` + `num_elements` (dynamically resizing the array if needed), and increment `num_elements`. To remove the next element, we find the element at position `start_index`. We delete the element here, and increment `start_index`. All of these operations take only constant time (with additions being amortized).

Notice that after k deletions, the first k spots in the array will be empty. We can save space by using these spots when `start_index` + `num_elements` exceeds the size of the array, instead of dynamically resizing (though we will still have to grow the array when `num_elements` exceeds the size of the array.

Problem 4 (10 Points).

(a) Show how to implement a stack using two queues. Analyze the running time of the stack operations.

(b) Show how to implement a queue using two stacks. Analyze the running time of the queue operations.

Solution:

(a) In a queue, we only have access to the oldest element, whereas in a stack, we need access to the newest element. The only way to get to the newest element in a queue is to remove all the other elements first. We will use a second queue to store the elements removed in this way. Specifically, we will have two queues: q_{main} and q_{aux}.

  - To add an element, simply add it to q_{main}. This takes constant time.
  - To remove an element, we need to get to the last element in q_{main}. Thus, we repeatedly remove all the elements of q_{main}, and put all but the last one into q_{aux}. The last element is the element we return. We then swap the pointers to q_{main} and q_{aux} so that q_{main} has all of the elements again. The running time of this operation is $O(n)$ because we need to process each of the elements.

(b) Similar to part (a), we will use two stacks s_{main} and s_{aux}.

  - To add an element, simply add it to s_{main}. This takes constant time.
  - To remove an element, we need to get to the bottom of the stack s_{main}. Thus, we repeatedly remove all the elements of s_{main}, and put all but the last one into s_{aux}. The last element is the element we return. We cannot just swap pointers in this case, as the elements of s_{aux} are reversed. Therefore, we must add all of the elements of s_{aux} back to s_{main} to get them in the right order.

Problem 5 (15 Points). We want hash functions to look like random functions, so why can't we just use a truly random function? Let $\mathcal{X}$ be a set with N elements, and $\mathcal{Y}$ a set with M elements.

(a) Show that the number of functions from $\mathcal{X}$ to $\mathcal{Y}$ is M^N.

(b) Suppose $\mathcal{X}$ is the set of n-bit sequences, and $\mathcal{Y}$ is the set of m-bit sequences. The random hash function needs to be written down somewhere. Show that the hash function can be represented using $m2^n$ bits.

(c) Show that this representation is tight: that any representation of all functions from $\mathcal{X}$ into $\mathcal{Y}$ must have at least one function that requires $m2^n$ bits. Assuming the number of subatomic particles in the visible universe is bounded by 10^{100}, and each subatomic particle can represent 1 bit, and that we are interested in hashing strings into single bits ($m = 1$), what length of sequences can we possibly hope to hash using truly random functions?

Solution:

(a) Let f be an arbitrary function from $\mathcal{X}$ to $\mathcal{Y}$. For each $x \in \mathcal{X}$, there are M possible values for the output $f(x)$. To choose f, we specify $f(x)$ for each x. Thus we are choosing one element out of M values a total of N different times, so there are M^N possibilities.

(b) Simply output the string that result from concatenating every output of f together. That is, output

$$f(0...00)f(0...01)f(0...10)...f(1...10)f(1...11) \ .$$

There are 2^n sets of m bits, so this representation requires $m2^n$ bits. To reconstruct f, set $f(x)$ to be the xth set of m bits, where x is interpreted as an integer, and we 0-index.

(c) Recall that k bits can represent at most 2^k different things. Thus, the number of functions that can be represented using a maximum of $m2^n - 1$ bits is

$$\sum_{k=0}^{m2^n-1} 2^k = 2^{m2^n} - 1 = M^N - 1$$

Therefore, at least 1 of the M^N different functions would require at least $m2^n$ bits to represent. Now we set $m = 1$, and want the largest n such that $2^n \leq 10^{100}$. Solving this equation gives $n = 332$. Thus, using all of the subatomic particles in the visible universe, we can hash strings of length at most 332 bits using a random function.

Problem 6 (15 Points). Suppose we want to create a hash table whose keys are student names (represented as a sequence of lower-case letters), and whose values are their GPAs. Let the size of the hash table be N. Explain why the following candidate functions are poor choices:

(a) Choose random integers A and B in $\{0, ..., N-1\}$. Define $h_1(s)$ as follows: decompose the string s into characters $s_1, ..., s_n$, and interpret these characters as integers, where a space is 0, "a" = 1, "b" = 2, etc. Then $h_1(s) = A(s_1 + ... + s_n) + B \mod N$.

(b) Using the same interpretation of characters as integers, $h_2(s) = s_n + 31s_{n-1} + 31^2 s_{n-2} + ... + 31^{n-1}s_1 \mod N$.

(c) $h_3(s) = t \mod N$ where t is the time, rounded to the nearest millisecond, when the function h_3 is called.

Solution:

(a) $h_1(s)$ has too many collisions. For example, if we let s' be a permutation of the characters of s, $h(s') = h(s)$. Therefore, a bad choice of inputs will result in many collisions and thus poor performance, even though $h_1(s)$ is randomized.

(b) $h_2(s)$ is similar to how strings are hashed in Java. If we look at $h_2(s)$ before the final modding, we see that since $s_i \leq 26 < 31$, each string is mapped to a unique output. However, $h_2(s)$ is not randomized, so there will be some sets of inputs that result in many collisions.

(c) If calls to h_3 come in at regular intervals, the outputs of h_3 may not be uniformly spread over $\{0, ..., N\}$. More importantly, $h_3(s)$ is not a function at all. Thus, when we put a string s into the table at time t_1, it will be put into slot $t_1 \mod N$. However, when we go to retrieve the GPA for s at a later time t_2, we will look at slot $t_2 \mod N$, which is most likely *not* the slot s was inserted into.

Problem 7 (10 Points). Prove the time bounds for a d-ary heap. That is, show that:

(a) `deletemin` operations take $O\left(\frac{d \log |V|}{\log d}\right)$.

(b) `insert` and `decreasekey` operations take $O\left(\frac{\log|V|}{\log d}\right)$.

Solution:

(a) To delete the minimum, we remove the root and replace it with the last node on the last row. If the root is larger than any of its children, we swap it with the smallest child and repeat. We perform at most h swaps, where h is the height of the tree. Since this is a complete tree where each node has d children, we perform $O(\log_d |V| = \log |V| / \log d)$ swaps, and for each swap, we need to compare to d children, giving a total running time of at most $O(d \log |V| / \log d)$

(b) To decrease the key, we check if the node is now less than its parent. If so, we swap and continue. Again, we perform h swaps, but we only need to compare to 1 parent, so the running time is $O(\log |V| / \log d)$. Since inserts just add the node to the end of the last row and decrease the key, inserts take the same amount of time

Problem 8 (15 Points). Show how to implement the following features on a balanced BST with n nodes:

(a) $\texttt{range}(r, a, b)$ takes as input the root r of the tree and two values a and b with $a \leq b$, and returns the sorted, possibly empty list of values between a (inclusive) and b (exclusive). If k values are output, your algorithm should run in time $O(k + \log n)$.

(b) $\texttt{numInRage}(r, a, b)$ counts the number of values between a (inclusive) and b (exclusive). Your algorithm should run in time $O(\log n)$, independent of the actual number of values in the range. This will require storing additional data in the tree. Explain how to modify insertions, deletions, and rotations to maintain the new data.

Solution:

(a) If r is null (i.e. empty tree), then the list of values is clearly empty. Otherwise, we can recursively find each of these ranges, and concatenate them:

$\texttt{range}(r, a, b) =$

- If r is empty, do nothing
- Otherwise, let L and R be the left and right children, and do:
  * If $r < a$, call $\texttt{range}(R, a, r)$
  * If $a \leq r < b$, call $\texttt{range}(L, a, r)$, report r, and call $\texttt{range}(R, r, b)$
  * If $r \geq b$, call $\texttt{range}(L, r, b)$

Correctness: as we mentioned above, if r is an empty tree, the range is empty. If $a > r$, then r and all nodes in the left subtree are not contained in the range, so we can ignore the left subtree. IF $a \leq r$, we need to report the values in the left subtree first since they are smaller, which must lie in the range $[a, r)$. If r is also less than b, then we need to report r next. Lastly, if r less than b, then there may be some values in the right subtree, and these values will be in the range $[r, b)$, so report these.

Running time: There is a constant number of operations per call to `range`, so to get the desired running time, we just need to bound the number of nodes we call `range` on that are not in the range (call these wasted nodes).

We will first consider a simplified case: all nodes in the tree are less than b. In this case, we claim that the number wasted nodes is at most h, the height of the tree. This is true if there is only one node. Further, if the root r is wasted, then it must be that $r < a$, meaning we only call `range` on the right child. By induction, the number of wasted nodes in the call to the right child is $h - 1$, so the total number of wasted nodes is h. If r is not wasted, then $r \geq a$, and the entire right subtree is contained in the range (no wasted nodes). The number of wasted nodes for the left child is $h - 1$ by induction, and thus the total number of wasted nodes is $h - 1$. The same proof holds if all nodes in the tree are greater than or equal to a.

Now for the general case, we claim that there are at most $2h - 1$ wasted nodes. First, if r is in the range, then the calls to both children are of the special case above, so by induction we get at most $2(h - 1)$ wasted nodes. If r is not in the range, say $r < a$. Then we only make a call to the right child, resulting in $2(h - 1) - 1$ wasted nodes. Adding the wasted node r results in $2h - 2$ wasted nodes. The same holds if $r \geq b$. Therefore, the total number of wasted nodes s at most $2h - 1 \in O(h) = O(\log n)$. Adding in the non-wasted nodes gives us a running time of $O(k + \log n)$.

(b) Store at each node r the total number of nodes in the subtree rooted at r, called $count(r)$. If r is not in the range, all the nodes in the range are in one subtree, so we just recursively get the count of that subtree. Otherwise, the number of nodes in the range is the total number in the tree, minus the number above and below. This gives us the following algorithm:

```
numInRange(r, a, b) =
  – If $r$ is empty, return 0
  – If $r < a$, let $R$ be the right child and call numInRange($R, a, b$)
  – If $r \geq b$, let $L$ be the left child and call numInRange($L, a, b$)
  – Otherwise, $a \leq r < b$. In this case:
      ∗ If $a = -\infty$, let $n_L = 0$
      ∗ Otherwise let $n_L$ = numInRange($L, -\infty, a$)
      ∗ If $b = \infty$, let $n_R = 0$
      ∗ Otherwise let $n_R$ = numInRange($R, b, \infty$)
      ∗ Return $count(r) - n_L - n_R$
```

Correctness: this algorithm is correct given the discussion above.

Running Time: First notice that each call to `numInRange` with a or b infinite results in exactly one recursive call. Further, if a and b are finite, there is either one recursive call, or two recursive calls each with an infinite boundary. Thus, for each level of the tree, there are at most two calls to `numInRange`, and there is a constant amount of work per call. Thus, the running time is $O(h) = O(\log n)$

Total points: 100