資訊工程系【演算法概論】期中考解答

1. Show that for any real constants a and b, where b > 0, $(n+a)^b = \Theta(n^b)$. (10%)

ANS:

To show that $(n+a)^b = \Theta(n^b)$, we want to find constants $c_1, c_2, n_0 > 0$ such that $0 \le c_1 n^b \le (n+a)^b \le c_2 n^b$ for all $n \ge n_0$.

Note that

$$\begin{aligned} n+a &\le n+|a| \\ &\le 2n \qquad \text{when } |a| \le n, \end{aligned}$$

and

$$\begin{aligned} n+a &\ge n-|a| \\ &\ge \frac{1}{2}n \qquad \text{when } |a| \le \tfrac{1}{2}n. \end{aligned}$$

Thus, when $n \ge 2|a|$,

$$0 \le \frac{1}{2}n \le n+a \le 2n.$$

Since $b > 0$, the inequality still holds when all parts are raised to the power b:

$$0 \le \left(\frac{1}{2}n\right)^b \le (n+a)^b \le (2n)^b,$$

$$0 \le \left(\frac{1}{2}\right)^b n^b \le (n+a)^b \le 2^b n^b.$$

Thus, $c_1 = (1/2)^b$, $c_2 = 2^b$, and $n_0 = 2|a|$ satisfy the definition.

2. Show that $2^{n+1} = O(2^n)$, but $2^{2n} \neq O(2^n)$. (10%)

ANS:

$2^{n+1} = O(2^n)$, but $2^{2n} \neq O(2^n)$.

To show that $2^{n+1} = O(2^n)$, we must find constants $c, n_0 > 0$ such that

$0 \leq 2^{n+1} \leq c \cdot 2^n$ for all $n \geq n_0$.

Since $2^{n+1} = 2 \cdot 2^n$ for all n, we can satisfy the definition with $c = 2$ and $n_0 = 1$.

To show that $2^{2n} \neq O(2^n)$, assume there exist constants $c, n_0 > 0$ such that

$0 \leq 2^{2n} \leq c \cdot 2^n$ for all $n \geq n_0$.

Then $2^{2n} = 2^n \cdot 2^n \leq c \cdot 2^n \Rightarrow 2^n \leq c$. But no constant is greater than all 2^n, and so the assumption leads to a contradiction.

3. Show that

(a) $T(n) = T(9n/10) + n = \Theta(n)$ (5%)

(b) $T(n) = 16T(n/4) + n^2 = \Theta(n^2 \lg n)$ (5%)

(c) $T(n) = 7T(n/2) + n^2 = \Theta\left(n^{\lg 7}\right)$ (5%)

(d) $T(n) = T(n-1) + n = \Theta(n^2)$ (5%)

(e) $T(n) = T\left(\sqrt{n}\right) + 1 = \Theta(\lg \lg n)$ (5%)

ANS:

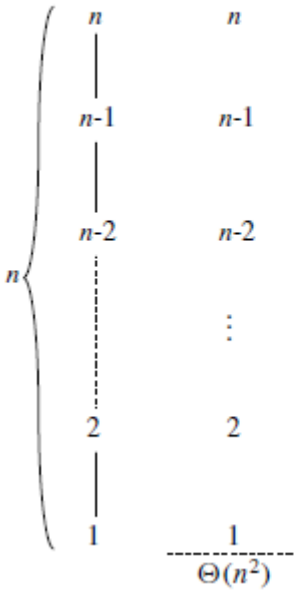
$T(n) = T(9n/10) + n = \Theta(n)$. This is a divide-and-conquer recurrence with $a = 1$, $b = 10/9$, $f(n) = n$, and $n^{\log_b a} = n^{\log_{10/9} 1} = n^0 = 1$. Since $n = \Omega(n^{\log_{10/9} 1+1})$ and $a/b^k = 1/(10/9)^1 = 9/10 < 1$, case 3 of the master theorem applies, and $T(n) = \Theta(n)$.

$T(n) = 16T(n/4) + n^2 = \Theta(n^2 \lg n)$. This is another divide-and-conquer recurrence with $a = 16$, $b = 4$, $f(n) = n^2$, and $n^{\log_b a} = n^{\log_4 16} = n^2$. Since $n^2 = \Theta(n^{\log_4 16})$, case 2 of the master theorem applies, and $T(n) = \Theta(n^2 \lg n)$.

$T(n) = 7T(n/2) + n^2 = O(n^{\lg 7})$. This is a divide-and-conquer recurrence with $a = 7$, $b = 2$, $f(n) = n^2$, and $n^{\log_b a} = n^{\log_2 7}$. Since $2 < \lg 7 < 3$, we have that $n^2 = O(n^{\log_2 7 - \epsilon})$ for some constant $\epsilon > 0$. Thus, case 1 of the master theorem applies, and $T(n) = \Theta(n^{\lg 7})$.

$T(n) = T(n-1) + n$

Using the recursion tree shown below, we get a guess of $T(n) = \Theta(n^2)$.



First, we prove the $T(n) = \Omega(n^2)$ part by induction. The inductive hypothesis is $T(n) \geq cn^2$ for some constant $c > 0$.

$$\begin{aligned} T(n) &= T(n-1) + n \\ &\geq c(n-1)^2 + n \\ &= cn^2 - 2cn + c + n \\ &\geq cn^2 \end{aligned}$$

if $-2cn + n + c \geq 0$ or, equivalently, $n(1-2c) + c \geq 0$. This condition holds when $n \geq 0$ and $0 < c \leq 1/2$.

For the upper bound, $T(n) = O(n^2)$, we use the inductive hypothesis that $T(n) \leq cn^2$ for some constant $c > 0$. By a similar derivation, we get that $T(n) \leq cn^2$ if $-2cn + n + c \leq 0$ or, equivalently, $n(1-2c) + c \leq 0$. This condition holds for $c = 1$ and $n \geq 1$.

Thus, $T(n) = \Omega(n^2)$ and $T(n) = O(n^2)$, so we conclude that $T(n) = \Theta(n^2)$.

$T(n) = T(\sqrt{n}) + 1$

The easy way to do this is with a change of variables, as on page 66 of the text. Let $m = \lg n$ and $S(m) = T(2^m)$. $T(2^m) = T(2^{m/2}) + 1$, so $S(m) = S(m/2) + 1$. Using the master theorem, $n^{\log_b a} = n^{\log_2 1} = n^0 = 1$ and $f(n) = 1$. Since $1 = \Theta(1)$, case 2 applies and $S(m) = \Theta(\lg m)$. Therefore, $T(n) = \Theta(\lg \lg n)$.

4. A d-ary heap is like a binary heap, but non-leaf nodes have d children instead of 2 children.
   (a) How would you represent a d-ary heap in an array? (5%)
   (b) What is the height of a d-ary heap of n elements in terms of n and d ? (5%)
   (c) Give an efficient implementation of EXTRACT-MAX in a d-ary max-heap. Analyze its running time in terms of d and n. (5%)
   (d) Give an efficient implementation of INSERT in a d-ary max-heap. Analyze its running time in terms of d and n. (5%)

ANS:

a. A d-ary heap can be represented in a 1-dimensional array as follows. The root is kept in $A[1]$, its d children are kept in order in $A[2]$ through $A[d+1]$, their children are kept in order in $A[d+2]$ through $A[d^2+d+1]$, and so on. The following two procedures map a node with index i to its parent and to its jth child (for $1 \le j \le d$), respectively.

D-ARY-PARENT(i)
return $\lfloor (i-2)/d + 1 \rfloor$

D-ARY-CHILD(i, j)
return $d(i-1) + j + 1$

To convince yourself that these procedures really work, verify that

D-ARY-PARENT(D-ARY-CHILD(i, j)) $= i$,

for any $1 \le j \le d$. Notice that the binary heap procedures are a special case of the above procedures when $d = 2$.

b. Please see the next page.

c. The procedure HEAP-EXTRACT-MAX given in the text for binary heaps works fine for d-ary heaps too. The change needed to support d-ary heaps is in MAX-HEAPIFY, which must compare the argument node to all d children instead of just 2 children. The running time of HEAP-EXTRACT-MAX is still the running time for MAX-HEAPIFY, but that now takes worst-case time proportional to the product of the height of the heap by the number of children examined at each node (at most d), namely $\Theta(d \log_d n) = \Theta(d \lg n / \lg d)$.

d. The procedure MAX-HEAP-INSERT given in the text for binary heaps works fine for d-ary heaps too. The worst-case running time is still $\Theta(h)$, where h is the height of the heap. (Since only parent pointers are followed, the number of children a node has is irrelevant.) For a d-ary heap, this is $\Theta(\log_d n) = \Theta(\lg n / \lg d)$.

b.

A path from the last node to the leaf would always be a longest path. This is because the last node is always at the lowest level in the heap. Now, assume the root is at level 0. Then the number of nodes at a completely filled level i would be d^i.

Let level k be the last completely filled level in the heap. So the number of nodes up to (and including) level k is:

$$\sum_{i=0}^{k} d^i = \frac{d^{k+1} - 1}{d - 1}$$

Now, the last node - the n^{th} node - can either be the last node at level k, or it can be in an incomplete level $k + 1$. Taking care of these two cases, it can be seen that:

$$\frac{d^{k+1} - 1}{d - 1} \leq n < \frac{d^{k+2} - 1}{d - 1}$$

$$\Rightarrow\ k \leq \log_d(n(d-1) + 1) - 1 < k + 1$$

Now, equality is only if the last node is the last leaf of level k, which also has distance k from the root. If not, that is if there is a level $k + 1$, then the log term would not be an integer, and applying the ceiling operator would give the right height of $k + 1$. Thus, if the last element of the array is at position n, the height of the heap is:

$$h = \lceil \log_d(nd - n + 1) \rceil - 1$$

5. Which of the following sorting algorithms are stable: insertion sort, merge sort, heapsort, and quicksort? (5%) Give a simple scheme that makes any sorting algorithm stable. (5%)

ANS:

Insertion sort is stable. When inserting $A[j]$ into the sorted sequence $A[1 \ldots j-1]$, we do it the following way: compare $A[j]$ to $A[i]$, starting with $i = j-1$ and going down to $i = 1$. Continue at long as $A[j] < A[i]$.

Merge sort as defined is stable, because when two elements compared are equal, the tie is broken by taking the element from array L which keeps them in the original order.

Heapsort and quicksort are not stable.

One scheme that makes a sorting algorithm stable is to store the index of each element (the element's place in the original ordering) with the element. When comparing two elements, compare them by their values and break ties by their indices.

6. In the selection algorithm SELECT, the input elements are divided into groups of 5. Will the algorithm work in linear time if they are divided into groups of 7? (5%) What is the minimum number of elements greater than x? (5%)

ANS:

For groups of 7, the algorithm still works in linear time. The number of elements greater than x (and similarly, the number less than x) is at least

$$4\left(\left\lceil\frac{1}{2}\left\lceil\frac{n}{7}\right\rceil\right\rceil-2\right)\geq\frac{2n}{7}-8\,,$$

7. Suppose that we use an open-addressed hash table of size m to store $n \leq m/2$ items. Assuming uniform hashing, show that for $i = 1, 2, \ldots, n$, the probability is at most 2^{-k} that the ith insertion requires strictly more than k probes. (15%)

ANS:

Since we assume uniform hashing, we can use the same observation as is used in Corollary 11.7: that inserting a key entails an unsuccessful search followed by placing the key into the first empty slot found. As in the proof of Theorem 11.6, if we let X be the random variable denoting the number of probes in an unsuccessful search, then $\Pr\{X \geq i\} \leq \alpha^{i-1}$. Since $n \leq m/2$, we have $\alpha \leq 1/2$. Letting $i = k + 1$, we have $\Pr\{X > k\} = \Pr\{X \geq k + 1\} \leq (1/2)^{(k+1)-1} = 2^{-k}$.