CS 435, Online Algorithms	**Homework 2 Recurrences and heaps**	Dr. David Nassimi

1. The following class of recurrences often arise in the analysis of divide-and-conquer algorithms. Note: a, b, c, d, β are all constants; and n is an integer power of b, $n = b^k$.)

$$T(n) = \begin{cases} a\,T(n/b) + c\,n^\beta, & n > 1 \\ d, & n = 1 \end{cases}$$

Let $h = \log_b a$. We used *repeated substitution* to derive the following solution, where A and B are some constants for each case.

$$T(n) = \begin{cases} A\,n^h + B\,n^\beta & = \Theta(n^\beta), & h < \beta \\ A\,n^h + B\,n^\beta & = \Theta(n^h), & h > \beta \\ A\,n^h + B\,n^h \log n & = \Theta(n^h \log n), & h = \beta \end{cases} \tag{1}$$

Use the above formula (Master Theorem) to find the solution form for each of the following recurrences, where n is a power of 2. Express the solution form invloving constants A and B. (Don't bother to find the constants.) Then express in $\Theta(\ )$ form.

(a) $T(n) = 2T(n/2) + 1$
(b) $T(n) = 2T(n/2) + n$
(c) $T(n) = 2T(n/2) + n^2$
(d) $T(n) = T(n/2) + 1$

2. Find the exact solution of the following recurrence, where n is a power of 2, $T(1) = 0$, and

$$T(n) = 4T(n/2) + n,\ n > 1.$$

(a) Use repeated substitution method.
(b) Find the solution form by using the above formula (Mater Theorem) and then find the constants A and B.

3. Mergesort algorithm is an efficient sorting algorithm bases on divide-and-conquer strategy. This algorithm sorts n elements, where $n > 1$, as follows:

- Divide the array into two halves. (If n is even, then each half has exactly $n/2$ elements. Otherwise, the two halves are of size $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$.)
- Sort each half recursively.
- Merge the two sorted halves.

Let $f(n)$ be the *worst-case number of key-comparisons* for mergesort algorithm to sort n elements.

Assume the special case when n is a power of 2. (That is, $n = 2^k$ for some integer k.) The following recurrence holds for this special case.

$$f(n) = \begin{cases} 2f(n/2) + n - 1, & n > 1 \\ 0, & n = 1. \end{cases} \tag{2}$$

Prove by induction the solution has the following upper bound. (Note: All logorithms in this course are assumed to be in base 2, unless stated otherwise.)

$$f(n) \le n \log n.$$

4. Consider the following recurrence (for the binary-search algorithm) where n is any integer.

$$f(n) = \begin{cases} 1, & n = 1 \\ f(\lfloor n/2 \rfloor) + 1, & n \geq 2. \end{cases}$$

Prove by induction that the solution is $f(n) = \lfloor \log_2 n \rfloor + 1$.

Hint: Observe that for any real number $r \geq 1$,

$$\lfloor \log \lfloor r \rfloor \rfloor = \lfloor \log r \rfloor.$$

Thus, for any integer $n \geq 2$,

$$\lfloor \log \lfloor n/2 \rfloor \rfloor = \lfloor \log(n/2) \rfloor = \lfloor \log n \rfloor - 1.$$

5. Consider the mergesort algorithm again. Let us now deal with the general case when n is any integer. Then, the worst-case number of key-comparisons is as follows.

$$f(n) = \begin{cases} f(\lceil n/2 \rceil) + f(\lfloor n/2 \rfloor) + n - 1, & n \geq 2 \\ 0, & n = 1. \end{cases} \tag{3}$$

Prove by induction that:

$$f(n) \leq n \lceil \log n \rceil.$$

Hint: Observe that

$$\lceil \log \lfloor n/2 \rfloor \rceil \leq \lceil \log \lceil n/2 \rceil \rceil.$$

Use the fact that for any integer $n \geq 2$,

$$\lceil \log \lceil n/2 \rceil \rceil = \lceil \log(n/2) \rceil = \lceil \log n \rceil - 1.$$

6. Let $A[1..n]$ be an existing min-heap of n elements. The operation DELETEMIN removes the smallest element and re-establishes the heap propety for the remaining $n - 1$ elements. The operation DELETEMIN is performed as follows:

```
TEMP = A[1]    //Retrieve the root, which is the smallest element.
A[1] = A[n]
n = n - 1
PUSHDOWN (A, 1, n)    // Pushdown the new root, A[1], to re-establish heap property.
return (TEMP)
```

Note that when the call PUSHDOWN$(A, 1, n)$ is made, the left and right subtrees of the root satisfy the heap property and the root needs to be pushed down as far as necessary (in the worst-case all the way to a leaf) to reestablish the heap property.

(a) Write the code for a *recursive* version of PUSHDOWN:

PUSHDOWN (dtype $A[\,]$, int r, int n)

where $A[1..n]$ is an array containing the heap of n elements to be re-established. The parameter r is the index of the root of the tree or subtree to be fixed (to be pushed down as far as it needs to go). Initially, the left and right subtrees of r already satisfy the heap property. At the end, the entire subtree rooted at r will satisfy the heap property.

(b) Write a *non-recursive* version of PUSHDOWN.

7. The heap algorithm BUILD-HEAP starts with an array $A[1..n]$ of n random elements and establishes a valid min-heap of n elements. In this problem, we will see how this may be done in $O(n)$ time using a divide-and-conquer approach. The algorithm is described recursively as follows. The parameter r in this recursive algorithm is the index of the root of the tree (or subtree) that needs to be built from scratch. The intial call to this algorithm is with $r = 1$, that is BUILD-HEAP$(A, 1, n)$.

```
BUILD-HEAP (dtype A[ ], int r, int n) {
if (2 * r > n) return;     //Return if r is a leaf node (or non-existing)
BUILD-HEAP(A, 2 * r, n)     //Build heap of left subtree
BUILD-HEAP(A, 2 * r + 1, n)     //Build heap of right subtree
PUSHDOWN (A, r, n)     //Now pushdown the root.
}
```

   (a) To analyze BUILD-HEAP, let $f(n)$ be the worst-case number of SWAP operations to build a heap of n elements, starting with the initial call BUILD-HEAP$(A, 1, n)$. To simplify the analysis,

   - Suppose that the tree is a full-binary-tree. So the number of nodes in the left and right subtrees each is $(n-1)/2 = \lfloor n/2 \rfloor < n/2$.
   - The worst-case number of swaps for PUSHDOWN$(A, 1, n)$ is $\lfloor \log n \rfloor \leq \log n$.

   To write a simplified recurrence for $f(n)$, we may drop the floor operations and use the upper bounds $n/2$ and $\log n$, respectively. Then,

$$f(n) = \begin{cases} 2f(n/2) + \log n, & n > 1 \\ 0, & n = 1. \end{cases}$$

   Prove by induction that the solution is $f(n) = An + B \log n + C$ and find the constants A, B, C.

   (b) Illustrate the recursive BUILD-HEAP algorithm on the following example:

$$A[1..13] = (13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1).$$

   Show the array in the form of a tree. Show the result after each PUSHDOWN.

   (c) Write a non-recursive version of BUILD-HEAP (with a series of calls to PUSHDOWN).

   (d) Illustrate the non-recursive BUILD-HEAP for the array $A[1..13] = (13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1)$. Show the intial tree, and the result after each PUSHDOWN is completed.

8. A priority queue (PQ) is an abstract data type (ADT) consisting of a set of elements of type record, where one of the fields is PRIORITY. The operations associated with this ADT are:

   - MAKENULL: to initialize an empty PQ;
   - INSERT: to add a new record;
   - DELETEMIN: to delete a record of lowest priority. (Our textbook calls this operation REMOVEMIN.)

   Consider an implementation of a PQ using a *quadary* heap (rather than the usual binary heap). In a quadary heap, each non-leaf node has 4 children (except possibly the last non-leaf node, which may have less than 4 children) and the priority of each node is $\leq$ those of its children.

   Let the nodes be numbered (indexed) 1 to n, in level-order, with the root as node 1. The heap is embeded in an array $A[1..n]$. Let the *levels* in the tree be 0 to h, with the root at level 0. (The numbering is illustrated below for $n = 16$. Priority values are not shown.)

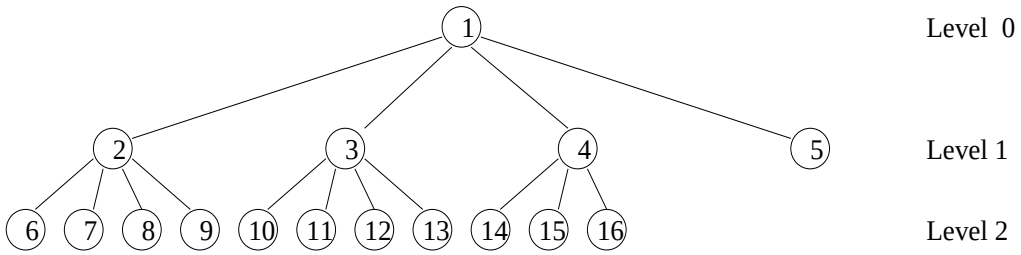
   (a) Find the indices of the four children of a node indexed i. Prove your answer by induction. Then, find the index of the parent of a node indexed i.

(b) Consider a FULL quadary heap of height h. (All nodes in levels 0 to $h-1$ have exactly 4 children, and all nodes at level h are leaves.) Find the total number of nodes, $N(h)$, as a function of h. Then, express the height h as a function of N.

(c) Now consider a COMPLETE quadary heap of n nodes with height h, where n is any integer. (Level h of the tree may not be full.) Find the height h as a function of n.
Hint: Let $N(h)$ be the function defined above. Observe that $N(h-1) < n \leq N(h)$.

(d) Write an efficient INSERT procedure in pseudo-code for adding a new record to a quadary heap. Analyze the worst-case number of key comparisons (exact number, not the order) in terms of n.

(e) Describe an efficient procedure (in general but precise way, without the programming details) for DELETEMIN operation on a quadary heap. Analyze the worst-case number of key comparisons (exact number, not order) in terms of n.

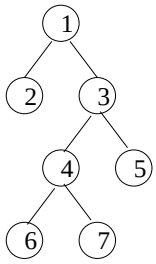


9. A binary tree is called *proper* if every non-leaf node has exactly two children. (An example is shown below.) Let n be the total number of nodes in a proper binary tree.

(a) What is the number of leaf nodes in terms of n? Prove your answer by induction.

(b) Write a recursive procedure (in pseudo-code) to find the maximum number of **left branches** found in any path from the root to a leaf. (In the example below, the path 1-3-4-6 has 2 left branches in it, which is the maximum value.)

(c) What is the time complexity of the above procedure? Explain.



10. This problem deals with computing x^n, where x is *real* and n is integer, using an efficient number of multiplications. More exactly, we want an efficient algorithm that uses $O(\log n)$ operations of only the following:

- real multiplications,
- integer add/subtract, integer multiply/divide by 2, integer mod 2,

and no other arithmetic operations.

(a) For the special case when n is a power of 2 ($n = 2^k,\ k \geq 1$), show how to compute x^n. What is the number of *real* multiplications?

(b) Write a recursive function, POWER(x,n), to compute x^n where n is an arbitrary integer ≥ 1. Let $f(n)$ be the worst-case number of real multiplications used. Write a recurrence for $f(n)$ and obtain the solution.

Additional Exercises (Not to be handed-in)

11. Consider the following sorting method, called 2-level selection sort.

    2.1 Let $n = p^2$ be the number of elements to be sorted. Find the largest element MAX as follows. Partition the elements into p sets, each of size p. Find the maximum element $M[i]$ of set i, for $1 \leq i \leq p$. Then find MAX as the maximum value of $M[i], 1 \leq i \leq p$. Output this MAX element (while deleting it from its set).

    2.2 Find the second largest element as follows. First update $M[i]$ for the set i from which MAX was taken, and then update MAX. Output MAX.

    2.3 Continue in a similar fashion for finding the third, fourth, etc.

    (a) What is the number of key comparisons (in terms of n) for finding the first MAX?

    (b) What is the worst-case number of key comparisons (in terms of n) for finding the second MAX?

    (c) Use (a) and (b) to find a reasonable *upper bound* on the worst-case number of key comparisons for the entire sorting algorithm.
    Hint: You may find a fairly good upper bound by observing that the worst-case for each subsequent MAX will not be any more than the worst-case for the second MAX.

    (d) Briefly describe an efficient way of modifying the above method to 3-level selection sort. (For each level, state the number of sets and the size of each set.) Assume n of convenient form to simplify your answers.

    (e) Analyze the worst-case number of key comparisons (in terms of n) for your 3-level selection sort by answering (a), (b), and (c) above.
    Compare the worst-case number of key comparisons for these two variations (2-level versus 3-level sort) and indicate which variation is faster.

12. (a) Starting with an empty AVL tree, insert the following sequence of elements into it.

    $$1, 2, 4, 10, 20, 5, 3, 6, 30, 40, 50$$

    Show the result after each insert. (When an insert operation makes the tree unbalanced, mark the nearest unbalanced ancestor and the type of rotation needed, and then the result after the tree is rebalanced.)

    (b) Delete 1, 2, 3. Show the result after each delete.

13. Consider a HASH table that uses linear-open-addressing as discussed in class. Suppose integer keys, a hash table of size 13, and a hash function $h(x) = x \bmod 13$. Assume there is only two marking for each table entry: either FREE or OCCUPIED.

    (a) Starting with an empty table, show the result after inserting the following sequence of elements into the hash table.

    $$5, 8, 4, 7, 2, 17, 20, 18, 21$$

    (b) Now delete 17. Show the sequence of key movements needed to fill the hole.