CS210: Data Structures and Algorithms	July-Nov 2013
Notes *2*	6 Aug, 2013

1 The divide-and-conquer paradigm: introduction

Divide-and-conquer is a general technique for algorithm design. Broadly, the technique breaks the problem into several sub-problems that are similar to the original problem, but smaller. These smaller problems are solved or "conquered", typically, using recursion, and the solutions are then combined to obtain a solution of the original problem. The combination step can sometimes be non-trivial. In this note, we look at an example illustrating this technique of algorithm design.

1.1 Merge-sort

The merge sort algorithm is an example of a sorting algorithm that follows the divide-and-conquer paradigm. The top-level routine is MERGE-SORT.

```
MERGE-SORT(A, p, r)
1.  if(p < r)   // returns if p ≥ r
2.      q = (p + r)/2   // Means ⌊(p + r)/2⌋
3.      MERGE-SORT(A, p, q)
4.      MERGE-SORT(A, q + 1, r)
5.      MERGE(A, p, q, r)
```

The algorithm works as follows. A is an array of length n. MERGE-SORT(A, p, r) is a call to sort the array $A[p \ldots r]$ (the end-indices p and r are inclusive). Line 1 is the test for the exit condition, namely, $p \geq r$. If this holds, then none of the other statements are executed and the function returns. Otherwise, we conceptually break the array into two about equal halves. Let q be a mid-point index, $q = \lfloor (p+r)/2 \rfloor$ and consider the two sub-arrays $A[p \ldots q]$ and $A[q+1 \ldots r]$. Why have we chosen equal halves? A study of recurrence equations will show that this minimizes the running time of such divide-and-conquer algorithms. Now, we set up the recursive steps. The array $A[p \ldots r]$ can be sorted if (a) $A[p \ldots q]$ is sorted, and (b) $A[q+1, \ldots r]$ is sorted, and, (c) after the recursive sorting operations, the two sorted sub-arrays $A[p \ldots q]$ and $A[q+1 \ldots r]$ are re-arranged (by the function MERGE) so that the entire array $A[p \ldots r]$ becomes sorted. The MERGE operation has to be designed so that it takes two sorted contiguous sub-arrays $A[p \ldots q]$ and $A[q+1 \ldots r]$ and returns $A[p \ldots r]$ in sorted order. A design of the MERGE function is shown below. It uses a simple COPY_with_sentinel(A, p, q, B) function that (a) copies the sub-array $A[p \ldots q]$ into the array $B[1 \ldots q-p+1]$, and, (b) inserts an ∞ into $B[q-p+2]$. (A sentinel literally means someone who is watching.)

```
COPY_with_sentinel(A, p, q, B)
1.  for  i = 1 to q − p + 1
2.      B[i] = A[p + i − 1]
```

3. $B[q-p+2] = \infty$

MERGE(A, p, q, r)
1. $n_1 = q - p + 1$
2. $n_2 = r - q$
3. Let $L[1 \ldots n_1 + 1]$ and $R[1 \ldots n_2 + 1]$ be new arrays
4. COPY_with_sentinel(A, p, q, L)
5. COPY_with_sentinel$(A, q + 1, r, R)$
6. $i = 1$
7. $j = 1$
8. **for** $k = p$ to r
9. **if**$(L[i] \leq R[j])$
10. $A[k] = L[i]$
11. $i = i + 1$
12. **else**
13. $A[k] = R[j]$
14. $j = j + 1$

The MERGE procedure works as follows. It assumes that the sub-arrays $A[p \ldots q]$ and $A[q+1 \ldots r]$ are each sorted. The sub-array $A[p \ldots q]$ is copied into an array $L[1 \ldots q-p+1]$ and the sub-array $A[q+1 \ldots r]$ is copied into an array $R[1 \ldots r-q]$. Also, we keep an additional slot in the arrays L and R that contains the special entry ∞, that is used as a sentinel (i.e., a guard value). This simplifies the description of the algorithm. [More on this later.] After the copying of the sub-arrays, the merging process is started, as follows. We can now think of the array $A[p \ldots r]$ as being empty. Keep two counters i for the array L and j for the array R. Compare $L[i]$ and $R[j]$, whichever is smaller, copy it into $A[k]$, and advance the corresponding counter. The for loop of line 8 always advances k after the loop. Notice the use of sentinel value ∞. If i becomes $q-p+2$ (or j becomes $r-q+1$) the value $L[i]$ is ∞ and therefore will not be the smaller value when $L[i]$ is compared with $R[j]$ for any value of $j = 1 \ldots r-q$. Also, when both $i = q-p+2$ and $j = r-q+1$, $k = (i-1)+(j-1)+p = r+1$, and the for loop has terminated [A loop invariant: At the beginning of each iteration of the for loop of line 8, $k = p + (i-1) + (j-1)$. Another way: at each iteration of the loop, one element is copied. The iteration ends when $r-p+1$ elements are copied. The first time a sentinel value is reached in either of the arrays, it is not copied since it is the larger (∞) value. Both sentinels are reached after $r-p+1$ elements are copied.]

Loop invariant: At the start of the for loop of lines 8–14, the subarray $A[p \ldots k-1]$ contains the $k-p$ smallest elements of $L[1 \ldots n_1+1] \cup R[1 \ldots n_2+1]$ in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied into A.

Analysis of Merge-Sort

Recursive programs are often analyzed by setting up a *recurrence equation* and then solving the recurrence equation. Let $T(n)$ be the running time of MERGE-SORT on an array of size n. For simplicity, assume that n is a power of 2, so that each division step yields two sub-problems of size exactly $n/2$. (We will later generalize this step). The time taken by MERGE is as follows. Lines 1-3 require constant time $\Theta(1)$. Lines 4 and 5 require time proportional to the length of the arrays

copied, which in total is $\Theta(n)$. The time taken by the for loop in lines 8-14 is proportional to n, since in each iteration of the for loop, exactly one element is copied, at the expense of some constant number of steps. Thus, the time required by MERGE for merging two sub-arrays of size $n/2$ each is bounded by some constant times n or $\Theta(n)$. Now consider MERGE-SORT. For an array of size n (n even), the division step requires constant time (statement 2). The next two statement are recursive calls to problems of size $n/2$ each. Hence, each of them require time $T(n/2)$, by definition of $T(\cdot)$. As we have seen, the function MERGE would require time $\Theta(n)$. This gives the recurrence equation

$$T(n) = \begin{cases} 2T(n/2) + \Theta(n) & \text{if } n \geq 2 \\ \Theta(1) & n = 1 \end{cases}$$

The exit condition for the recursion is when the length of the array is 1, this is given by line 1 in function MERGE-SORT, and takes $\Theta(1)$ time. Write the above recurrence as follows.

$$T(n) = \begin{cases} 2T(n/2) + cn & \text{if } n \geq 2 \\ c & n = 1 \end{cases} \tag{1}$$

Let c be the constant representing the time required to solve problems of size 1 as well as the time per array element for problem division and combining the solutions (merge). We will see a general *master theorem* for solving such recurrences. However, now we consider the recursion tree approach (and it is convenient to consider n to be a power of 2 for this purpose). Figure 1 shows the recursion tree of recurrence equation of Equation (1). Part (a) of the figure shows a one-level unrolling of the recursion (assuming $n \geq 2$). The same is unrolled one level further in part (b). Part (c) shows a general tree for n equal to some power of 2. Here the nodes are expanded until each leaf node is a problem of size 1, having cost c.

Next, (and this is an interesting step with recursion trees), we add the costs of each level of the tree. The top level contributes cn. The next level contributes a sum of $c(n/2) + c(n/2) = cn$. The level below that would have 4 nodes, each having cost $c(n/4)$, to give a total cost of cn. In this way, the set of nodes at level l incur a total cost of cn. Since there are $1 + \log n$ levels (including the root), the total cost is $T(n) = cn\log(n) + cn$. Thus, $T(n) = \Theta(n\log(n))$.

Analysis revisited. In the analysis of the MERGE function, we made the assumption that creating the temporary arrays L and R had a cost of 1. First, we note that the use of sentinels is only for convenience. Solve the following exercise.

Exercise 2.1

1. Rewrite the MERGE function so that it does not use the sentinels.
2. Now write a merge function MERGE(A, p, q, r, B) that assumes that the sub-arrays $A[p \ldots q]$ and $A[q \ldots r]$ are sorted, and uses only the empty sub-array $B[p \ldots r]$ for intermediate computation. The array $B[1 \ldots n]$ is created in the call to MERGE-SORT.

Exercise 2.2 Problem 2.3-7 of [CLRS]. Describe a $\Theta(n\log(n))$-time algorithm that, given an array consisting of n integer elements and another integer x, determines whether or not there exist two element in S whose sum is exactly x.

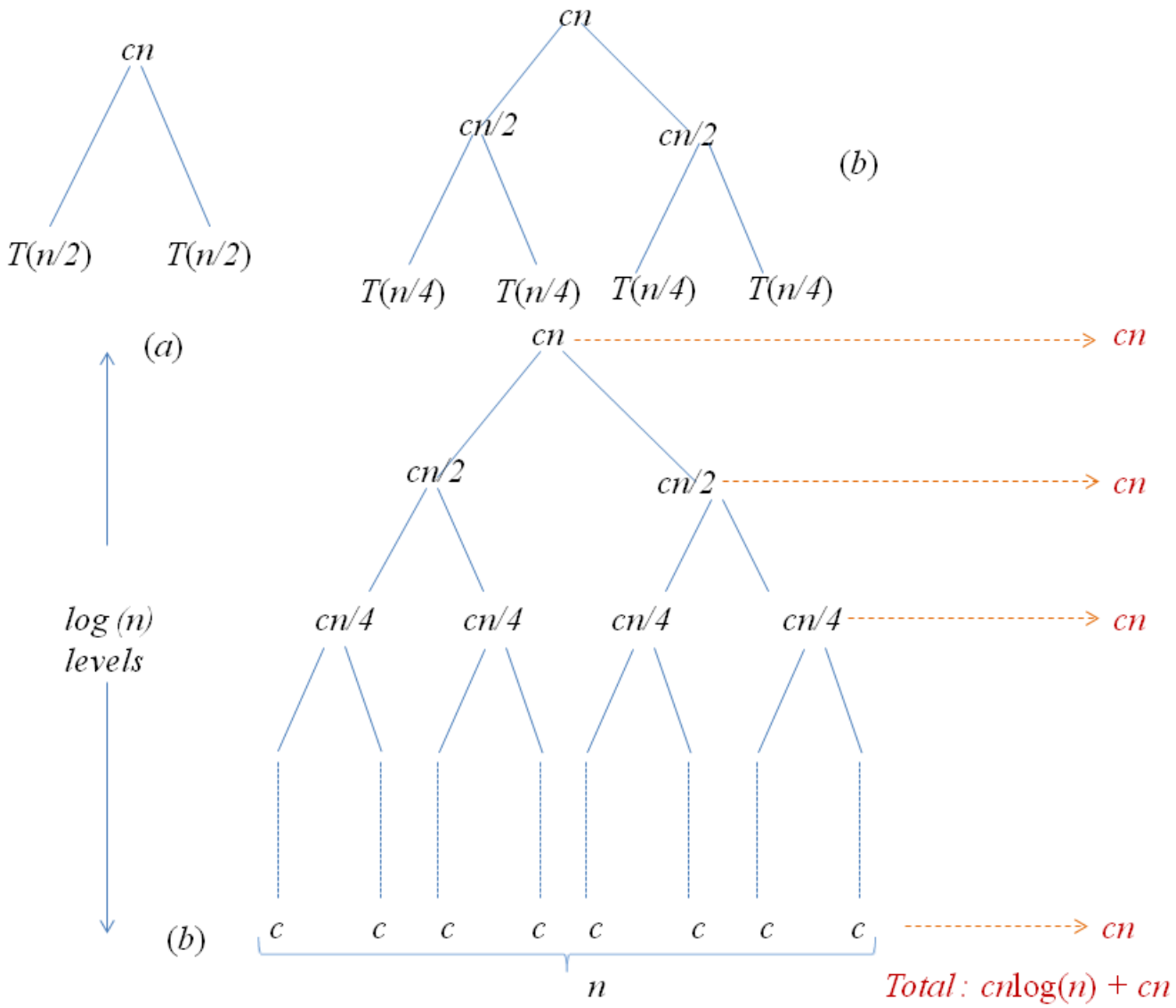

Figure 1: Recursion tree for solving the recurrence equation: $T(n) = 2T(n/2) + cn, T(1) = 1$.