Chapter 2.3: Designing Algorithms and Merge sort

Techniques for Designing Algorithms

- Insertion sort, selection sort, linear search use an incremental algorithm design techniques. These usually result in *iterative* algorithms.
- Recursive algorithms are useful and common and provide a different way of tackling problems. It goes hand-in-hand with the divide-and-conquer algorithm design technique.
- Divide-and-Conquer is an useful recursive technique for designing algorithms. It consists of three steps:

  Divide the problem into a number of subproblems that are smaller instances of the same problem.

  Conquer the subproblems by solving them recursively. If the subproblems are small enough, just solve the problems directly.

  Combine the solutions of the subproblems into the solution for the original problem.

Merge sort: a divide and conquer algorithm

- **Merge sort** is an example of a divide-and-conquer algorithm.

  Divide: the n-element sequence to be sorted into two subsequences of $n/2$ elements each.

  Conquer: sort the two subsequences recursively using merge sort.

  Combine: by merging the two sorted subsequences to produce a sorted answer.

- The base case for the recursion is when the sequence to be sorted has length 1. An array with one element is already sorted.
- The key part is the combine step that merges to two sorted subsequence. This requires a separate output array. The merge algorithm walks through the two sorted subsequences, copying the smaller value to the output array. If one subsequence is exhausted, then we simply copy the remaining elements of the other subsequence to the output array. Let's examine the pseudo-code for merge procedure shown below.

```
MERGE(A, p, q, r)
1  n_L = q − p + 1          // length of A[p : q]
2  n_R = r − q              // length of A[q + 1 : r]
3  let L[0 : n_L − 1] and R[0 : n_R − 1] be new arrays
4  for i = 0 to n_L − 1     // copy A[p : q] into L[0 : n_L − 1]
5      L[i] = A[p + i]
6  for j = 0 to n_R − 1     // copy A[q + 1 : r] into R[0 : n_R − 1]
7      R[j] = A[q + j + 1]
8  i = 0                    // i indexes the smallest remaining element in L
9  j = 0                    // j indexes the smallest remaining element in R
10 k = p                    // k indexes the location in A to fill
11 // As long as each of the arrays L and R contains an unmerged element,
   //     copy the smallest unmerged element back into A[p : r].
12 while i < n_L and j < n_R
13     if L[i] ≤ R[j]
14         A[k] = L[i]
15         i = i + 1
16     else A[k] = R[j]
17         j = j + 1
18     k = k + 1
19 // Having gone through one of L and R entirely, copy the
   //     remainder of the other to the end of A[p : r].
20 while i < n_L
21     A[k] = L[i]
22     i = i + 1
23     k = k + 1
24 while j < n_R
25     A[k] = R[j]
26     j = j + 1
27     k = k + 1
```

- Observations about the Merge procedure
  - The algorithm is **oblivious** in the sense that its run-time is the same for different inputs. However, it won't be considered **oblivious** if we define it to mean that the control flow for different inputs to be the same.
  - The worst-case run-time is $\Theta(n)$. See below for the analysis.
  - Note the number of elements in the left subsequence is n_L and the the number of elements in the right subsequence is n_R

```
MERGE(A, p, q, r)
 1  n_L = q − p + 1          // length of A[p : q]
 2  n_R = r − q              // length of A[q + 1 : r]
 3  let L[0 : n_L − 1] and R[0 : n_R − 1] be new arrays
 4  for i = 0 to n_L − 1     // copy A[p : q] into L[0 : n_L − 1]
 5      L[i] = A[p + i]
 6  for j = 0 to n_R − 1     // copy A[q + 1 : r] into R[0 : n_R − 1]
 7      R[j] = A[q + j + 1]
 8  i = 0                    // i indexes the smallest remaining element in L
 9  j = 0                    // j indexes the smallest remaining element in R
10  k = p                    // k indexes the location in A to fill
11  // As long as each of the arrays L and R contains an unmerged element,
    //      copy the smallest unmerged element back into A[p : r].
12  while i < n_L and j < n_R
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
18      k = k + 1
19  // Having gone through one of L and R entirely, copy the
    //      remainder of the other to the end of A[p : r].
20  while i < n_L
21      A[k] = L[i]
22      i = i + 1
23      k = k + 1
24  while j < n_R
25      A[k] = R[j]
26      j = j + 1
27      k = k + 1
```

$n = n_L + n_R$

$\Theta(1)$

$\Theta(n_L + n_R)$

$\Theta(n_L)$

$\Theta(n_R)$

$\Theta(1)$

$\Theta(\max(n_L, n_R)) \le \Theta(n)$

$\Theta(1)$

$\Theta(n_L)$

$\Theta(1)$

$\Theta(n_R)$

$\Theta(1)$

Total runtime $= \Theta(1) + \Theta(n_L + n_R) + \Theta(n_L) + \Theta(n_R) + \Theta(1) + \Theta(n) + \Theta(n_L) + \Theta(n_R)$

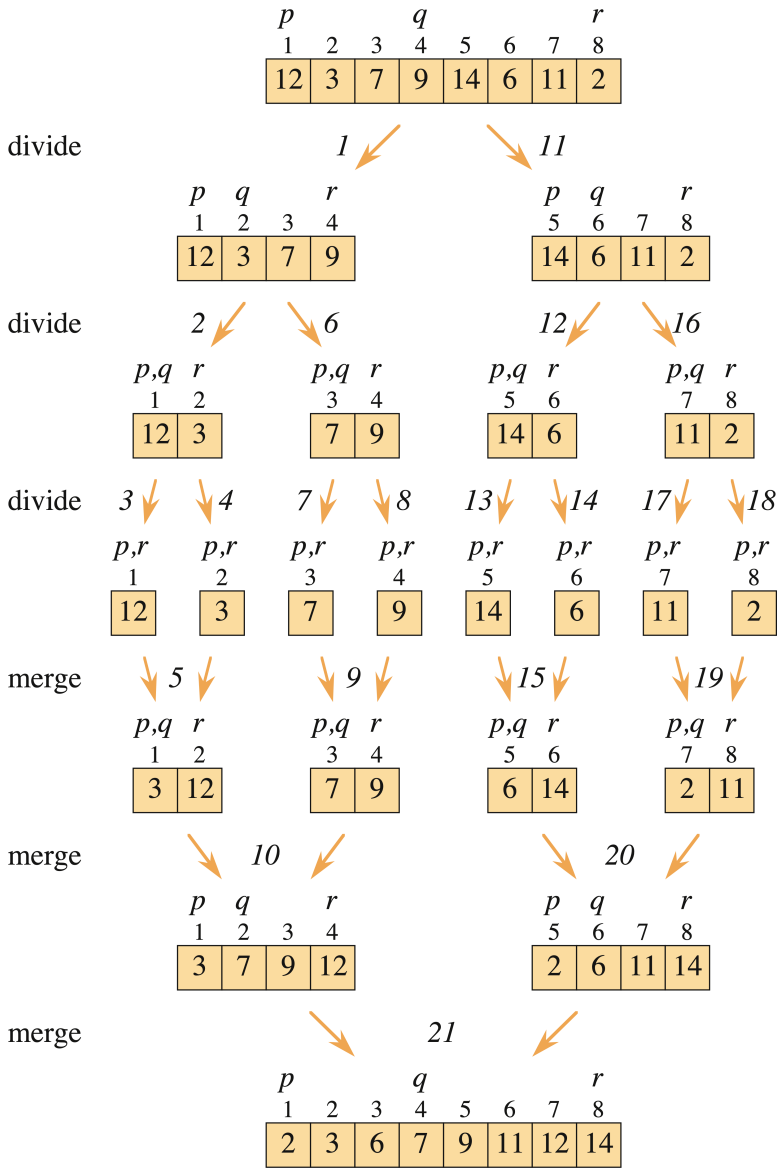
$= \Theta(1 + n_L + n_R + n_L + n_R + 1 + n + n_L + n_R)$

$= \Theta(1 + n + n + 1 + n + n) = \Theta(4n + 2) = \Theta(n)$ ✓

- Now we can write out the pseudo-code for the merge sort algorithm:

```
MERGE-SORT(A, p, r)
1  if p ≥ r                          // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋                   // midpoint of A[p : r]
4  MERGE-SORT(A, p, q)               // recursively sort A[p : q]
5  MERGE-SORT(A, q + 1, r)           // recursively sort A[q + 1 : r]
6  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
7  MERGE(A, p, q, r)
```

- Here is an example of running merge sort on the input:



- Try Exercise 2.3-1 (on your own): Try merge sort on the input $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.

Analysis of Divide and Conquer Algorithms

- The run time of a divide-and-conquer algorithm can be described by a recurrence equation (or just recurrence). Here is the general form:

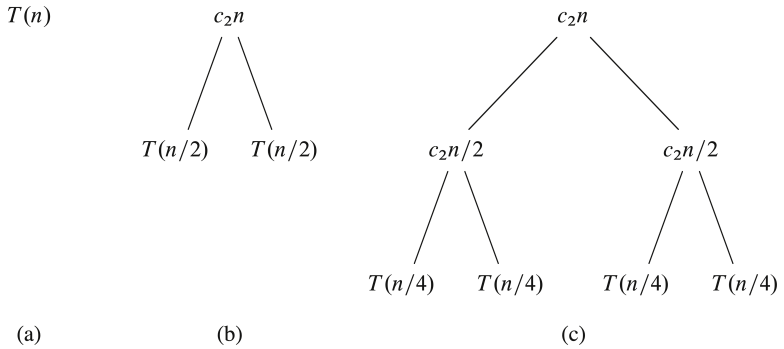
$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c, \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

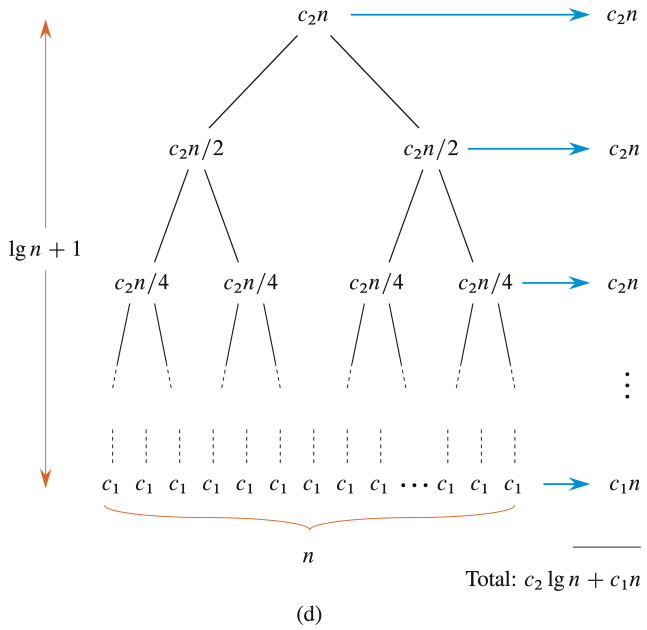
- For merge sort, a = 2, b = 2, and $c = 1$. Also, c_1 and c_2 are constants that depend on the merge and mergesort code.

$$T(n) = \begin{cases} c_1 & \text{if } n \leq 1, \\ 2T(n/2) + c_2 n & \text{otherwise} \end{cases}$$

- Techniques for solving recurrence equations:
  - *guess* and then prove by mathematical induction
  - *substitution method* $\rightarrow$ keep expanding the recurrence until we can come up with the closed form. To be thorough, we would also have to prove using induction that the closed form is correct, although we usually skip this step when it is straightforward.
  - *draw a recurrence tree* and then use the tree to add up the run-time. I find this the more intuitive method.

- Running time analysis, by drawing the recurrence tree:





- Note that at level 1, we are merging two lists of total size n, which takes time $c_2 n$.
- Note that at level 2, we are pairwise merging two lists of size $n/2$ each, which takes time $c_2 \frac{n}{2} \times 2 = c_2 n$.
- Note that at level 3, we are pairwise merging four lists of size $n/2^2 = n/4$ each, which takes time $c_2 \frac{n}{4} \times 4 = c_2 n$.

  ...
- Note that at level k, we are pairwise merging 2^k lists of size $n/2^k$ each, which takes time $c_2 \frac{n}{2^k} \times 2^k = c_2 n$.
- The recursion stops when $n/2^k = 1 \rightarrow k = \lg n$. This assumes that n is a power of 2. If it isn't the number of levels is the next integer higher than $\lg n$, which we denote by the ceiling function $\lceil \lg n \rceil$.

- When the recursion stops, we have n subsequences of length 1 each (the base case). These take $c_1 n$ (the base case).
- Thus, total running time is $\Theta(c_2 n \lg n + c_1 n) = \Theta(n \lg n)$.

- **Recommended Exercises**:
  - Ex 2.3-6: Binary Search (See notes on Searching for an answer)
  - Ex 2.3-7. Insertion sort combined with binary search
  - Problem 2.1: Merge sort combined with insertion sort

- **Extra reading (or listening!)**: Check out the song "*There's a hole in my bucket*" for recursion without a base case! It even has a Wikipedia page for it: https://en.wikipedia.org/wiki/There%27s_a_Hole_in_My_Bucket

  Here is a classic performance of the song!

  https://www.youtube.com/watch?v=xVAvMIhvqfk&ab_channel=HarryBelafonteTe