Runtime Complexity of Algorithm

Paramita Koley

Post Doctoral Fellow, ISI

Runtime complexity of algorithm

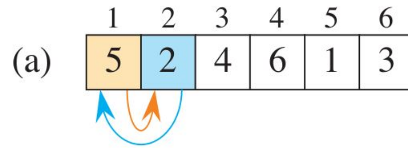
- Assumptions regarding execution
  - instructions execute one after another, with no concurrent operations
  - each instruction takes the same amount of time as any other instruction and that each data access for using the value of a variable or storing into a variable takes the same amount of time as any other data access
- Running time of an algorithm is measured by the number of instructions and data accesses executed
  - depends on instructions, number of data access, and many factors.. Most importantly size of input data
- Input data depends on problem
  - Sorting, searching algorithms – Input size: number of elements in the array
  - DFS, BFS, etc graph algorithms. Input size: Graph (No of vertices and no of edges)
  - Multiply two integer. Input size = No of bits required to represent the numbers

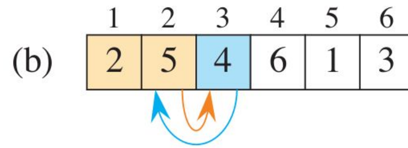
A sample algorithm

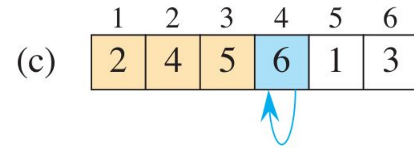
- Problem: Given an array of elements, sort the elements of the array
- Consider a simple algorithm

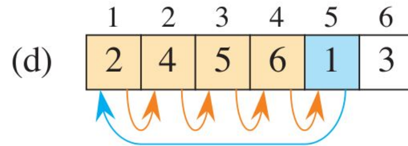
```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i - 1].
4      j = i - 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j - 1
8      A[j + 1] = key
```

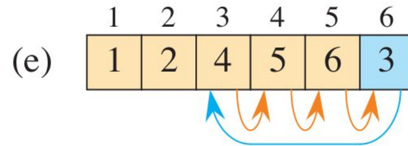
Insertion sort

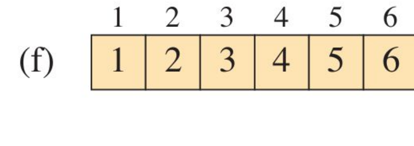












Runtime analysis of Insertion sort

INSERTION-SORT(A, n)	*cost*	*times*
1 **for** $i = 2$ **to** n	c_1	n
2 $key = A[i]$	c_2	$n - 1$
3 // Insert $A[i]$ into the sorted subarray $A[1 : i - 1]$.	0	$n - 1$
4 $j = i - 1$	c_4	$n - 1$
5 **while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
6 $A[j + 1] = A[j]$	c_6	$\sum_{i=2}^{n} (t_i - 1)$
7 $j = j - 1$	c_7	$\sum_{i=2}^{n} (t_i - 1)$
8 $A[j + 1] = key$	c_8	$n - 1$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7 \sum_{i=2}^{n} (t_i - 1) + c_8(n-1) .$$

Best case: the input array is sorted

$$t_i = 1$$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \;. \end{aligned}$$

Run time is linear function of n

Worst case: the input array is sorted in decreasing order

$$t_i = i$$

$$\sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

$$\sum_{i=2}^{n} i = \left(\sum_{i=1}^{n} i\right) - 1 = \frac{n(n+1)}{2} - 1$$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) \,. \end{aligned}$$

Run time is function of n^2

Why worst case running time

- Provides an reasonable upper bound on the running time for any input
- Sometimes the worst case occurs fairly often
  - Example, searching database for a item not present
- Average case is often roughly as bad as the worst case
  - Input: an array of n randomly chosen numbers
  - Where in subarray A[1:i-1] to insert element A[i]
  - On average, half the elements in A[0:i-1] are less than A[i]
  - On average, therefore, A[i] is compared with just half of the subarray, i/2 elements
  - The resulting average-case running time turns out to be a quadratic function of the input size

Order of growth

Order of growth

- We are not interested in actual cost of the running time
- We ignore the abstracts of the expression
- We are actually interested in order of growth of the running time, thereby we ignore even the lower-order terms or coefficient of the highest order term
- We use greek letter Θ (Theta) to denote the order of growth
- $\Theta(n^2)$ means roughly proportional to n^2 when n is large
- $\Theta(n)$ means roughly proportional to n when n is large

$$T(n) = an^2 + bn + c \; = \; \Theta(n^2)$$

$$T(n) = an + b \; = \; \Theta(n)$$

Divide and conquer

Divide and conquer algorithms

- Insertion sort is an example of incremental algorithm
- Now we will discuss another algorithm design paradigm - divide and conquer approach
    - Advantage: Run-time complexity analysis is often straight-forward

Divide and conquer approach

- Divide the original problem into one or more smaller sub-problems
- Conquer each of the subproblem by either directly solving it (if it is small enough) or recursively solving it
- Combine the subproblem solutions to form a solution of the original problem

Example: Merge sort

Merge Sort

- Divide the subarray A[p:r] to be sorted into two adjacent subarrays, each of half the size
  - Compute the midpoint q of A[p:r] (taking the average of p and r)
  - Divide A[p:r] into subarrays A[p:q] and A[q+1:r]
- Conquer by sorting each of the two subarrays A[p:q] and A[q+1:r] recursively using merge sort.
- Combine by merging the two sorted subarrays A[p:q] and A[q+1:r] back into A[p:r], producing the sorted answer.

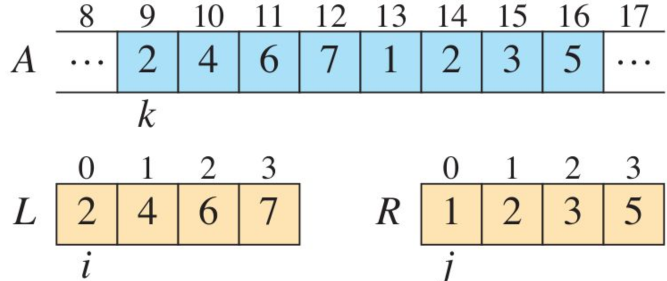
Combine Step

- It assumes as input A with two sorted subarrays A[p:q] and A[q+1:r].
- Copy the subarrays to two separate arrays L and R
- It chooses the smaller element of L and R and copy it back to A
- Repeat this step until one subarray is empty
- We call this subroutine as **Merge**

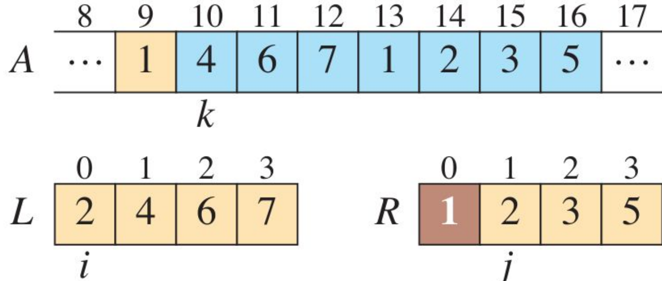
```
MERGE(A, p, q, r)
1   n_L = q - p + 1                  // length of A[p:q]
2   n_R = r - q                      // length of A[q+1:r]
3   let L[0:n_L - 1] and R[0:n_R - 1] be new arrays
4   for i = 0 to n_L - 1             // copy A[p:q] into L[0:n_L - 1]
5       L[i] = A[p + i]
6   for j = 0 to 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
```

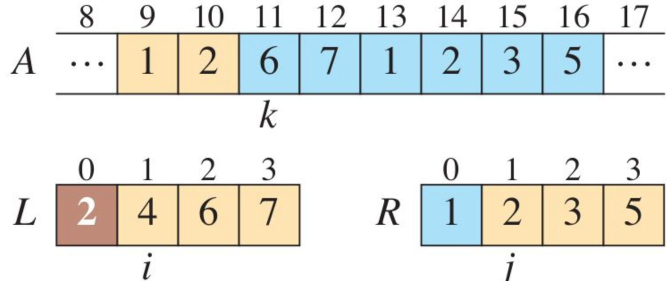
Details of Merge



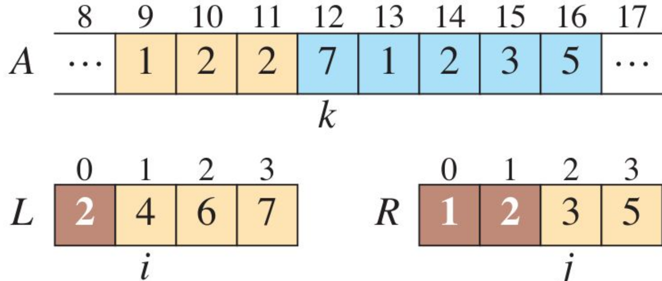
(a)



(b)

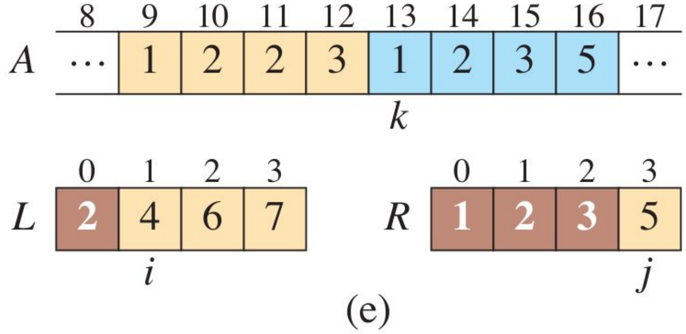


(c)

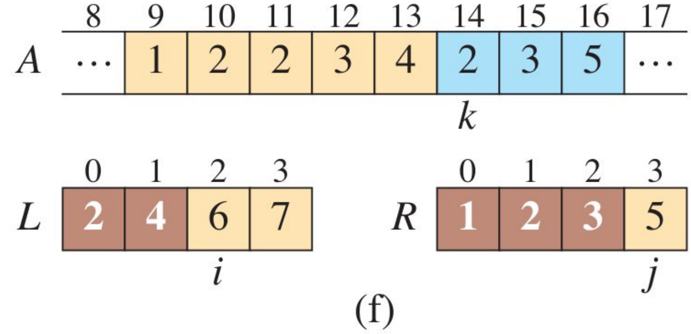


(d)

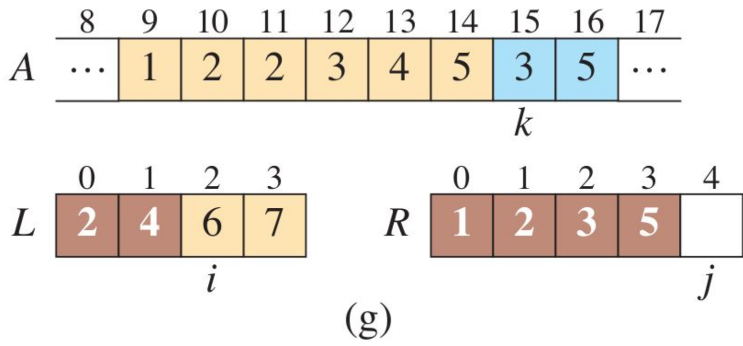
Details of Merge



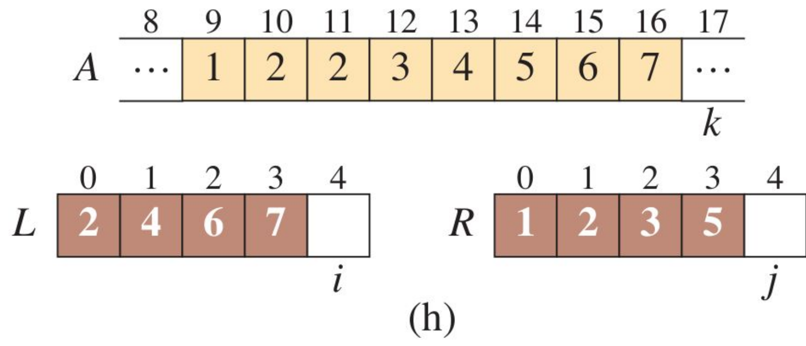
(e)



(f)



(g)



(h)

Time complexity of Merge

- Comparing top of both stacks and copying the smallest to array A takes $\Theta(1)$
- Once one of the stack is empty, rest elements of the other stack are copied to A
- If all elements of one stack < all elements of the other stack, at least n/2 comparisons to be made, after that n/2 copy operations are to be made
- If elements are copied from alternative array, n-1 comparisons are to be made

Precisely $\Theta(n)$

Merge-sort procedure using merge as subroutine

```
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)
```

Run time analysis of general divide and conquer algorithm

Let divide step breaks the problem into “a” subproblems, each of size b

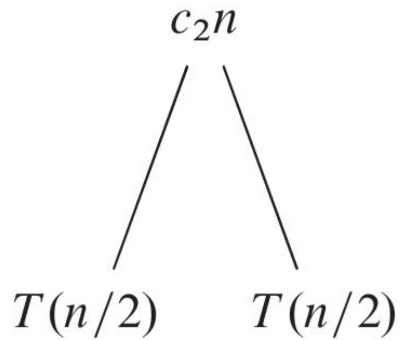
$$T(n) = \begin{cases} \Theta(1) & \text{if } n < n_0 , \\ D(n) + aT(n/b) + C(n) & \text{otherwise} . \end{cases}$$

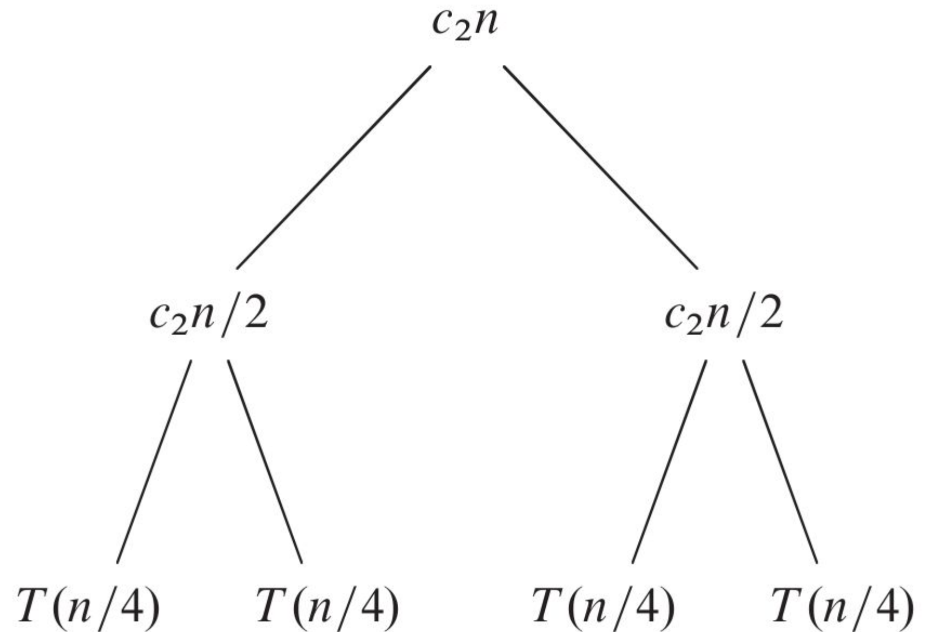
For merge sort, D(n) = $\Theta(1)$, C(n) = $\Theta(n)$, a=b=2

$$T(n) = 2T(n/2) + \Theta(n) .$$

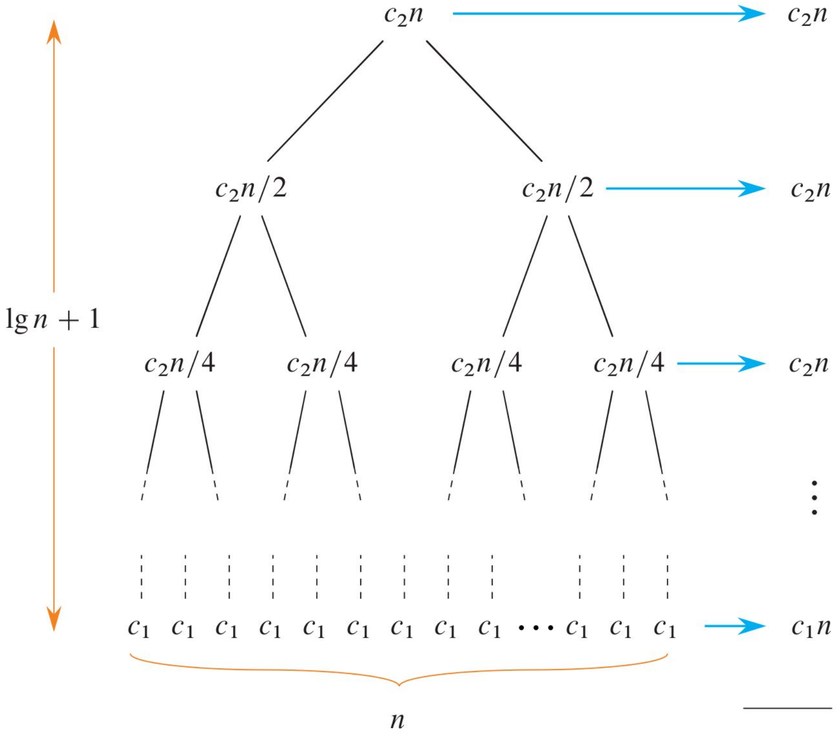
Solving recurrence relation via recurrence tree

$T(n)$





Recursion Tree



Total cost =

$$c_2 n \lg n + c_1 n = \Theta(n \lg n)$$

Asymptotic efficiency

- Asymptotic efficiency: we are concerned with how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound
- Usually, an algorithm that is asymptotically more efficient is the best choice for all but very small inputs

Types of Asymptotic Bound

Asymptotic Upper Bound

$$O(n^2)$$

Asymptotic Lower Bound

$$\Omega(n^2)$$

Asymptotic Tight Bound

$$\Theta(n^2)$$

End