

Divide and Conquer

Algorithm D-and-C(n : input size)

```
if  $n \leq n_0$  /* small size problem */  
    Solve problem without further sub-division;  
else  
    Divide into  $m$  sub-problems;  
    Conquer the sub-problems by solving them  
        independently and recursively; /* D-and-C( $n/k$ ) */  
    Combine the solutions;
```

Advantage: straightforward and running times are often easily determined.

Complexity of Divide and Conquer

Suppose we divide the original problem into m sub-problems, each with a problem size n/k , and let $\mathbf{D(n)}$ be the time needed to do the dividing, and $\mathbf{C(n)}$ the time needed to do the combining. Then we got a very general formula to compute $\mathbf{T(n)}$:

$$T(n) = \begin{cases} \Theta(1) & , \text{ for small sizes} \\ mT(n/k) + D(n) + C(n) & , \text{ otherwise} \end{cases}$$

Merge Sort

```
MERGE-SORT(A, p, r) {  
  if (p == r)          /* small size */  
    return;  
  else  
    q = (p + r)/2;  
    Merge-Sort(A, p, q);  
    Merge-Sort(A, q+1, r);  
    Merge(A, p, q, r);  
}
```

To sort the whole array, Merge-Sort(A, 1, n) is called.

Complexity of Merge Sort

Divide: The divide step only computes the middle, takes only constant time. $D(n) = \Theta(1)$.

Conquer: Recursively sort 2 subarrays. $C(n) = 2T(n/2)$.

Combine: Merge two $n/2$ -element subarrays, in linear time $\Theta(n)$.

Overall:

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

Based on Master Method-Case 2,

$$T(n) = \Theta(n \lg n)$$

for best, worst and average cases.

Quick Sort

Basic Steps:

- *Divide*: split the array $A[p..r]$ into two nonempty subarrays $A[p..q]$ and $A[q+1..r]$ such that each element of $A[p..q]$ is *less than or equal* to each element of $A[q+1..r]$.
- *Conquer*: Sort the two subarrays by calling quicksort recursively.
- *Combine*: Trivial.

Quick Sort algorithm

```
QUICK-SORT(A, p, r)
  if (p == r)          /* small size problem*/
    return;
  else
    q := Partition(A, p, r);
    Quick-Sort(A, p, q-1);
    Quick-Sort(A, q+1, r);
```

To sort the whole array, Quick-Sort(A, 1, n) is called.

Partition

- There are several different strategies that can be used by **Partition**.
- One possible strategy:
as we scan from left to right, we move the left bound to the right when the element is less than the pivot, otherwise we swap it with the rightmost unexplored element and move the right bound one step closer to the left.
- Note: although the strategy used in CLRS textbook takes different form, it is essentially the same as the above.

Illustration of Partition

Pivot about the last element: 10. We keep 3 sections: the elements ≤ 10 , elements > 10 and the unexplored elements.

```
| 17 12 6 19 23 8 5 || 10
| 5 12 6 19 23 8 | 17 || 10
5 | 12 6 19 23 8 | 17 || 10
5 | 8 6 19 23 | 12 17 || 10
5 8 | 6 19 23 | 12 17 || 10
5 8 6 | 19 23 | 12 17 || 10
5 8 6 | 23 | 19 12 17 || 10
5 8 6 || 23 19 12 17 || 10
5 8 6 | 10 | 19 12 17 23
```

Complexity: consists of $n - 1$ comparisons and at most n swaps. So $T(\text{Partition}) = \Theta(n)$.

Complexity of Quick Sort

Divide: The divide step **Partition** takes linear time $\Theta(n)$.

Conquer: Recursively sort 2 subarrays, one with size $q - 1$, the other is $n - q$. $C(n) = T(q - 1) + T(n - q)$.

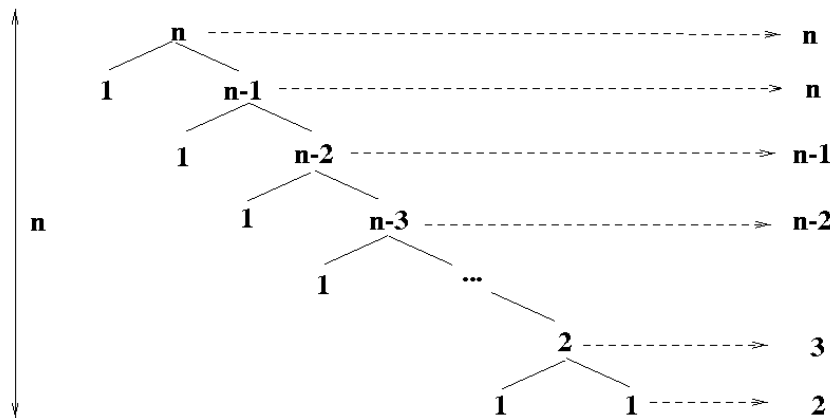
Combine: Basically no Combine step.

$$T(n) = T(q - 1) + T(n - q) + \Theta(n).$$

Worst Case

Partition always separates the array into one 0-length and one $(n - 1)$ -length subarrays. If we pick the last element as the pivot, this situation happens when the input is sorted ascendingly or descendingly.

- **Worst case splitting**

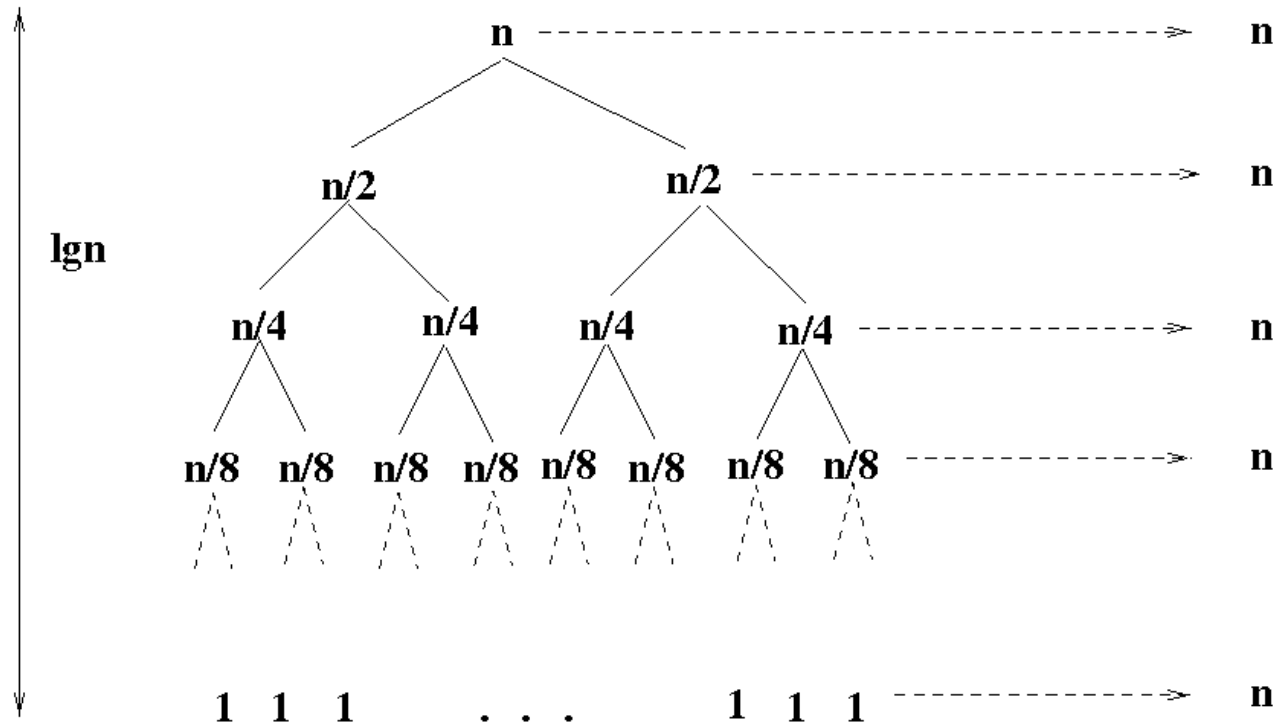


$$T(n) = T(n - 1) + \Theta(n)$$

$$T(n) = \Theta(n^2)$$

Best Case

- Best case splitting

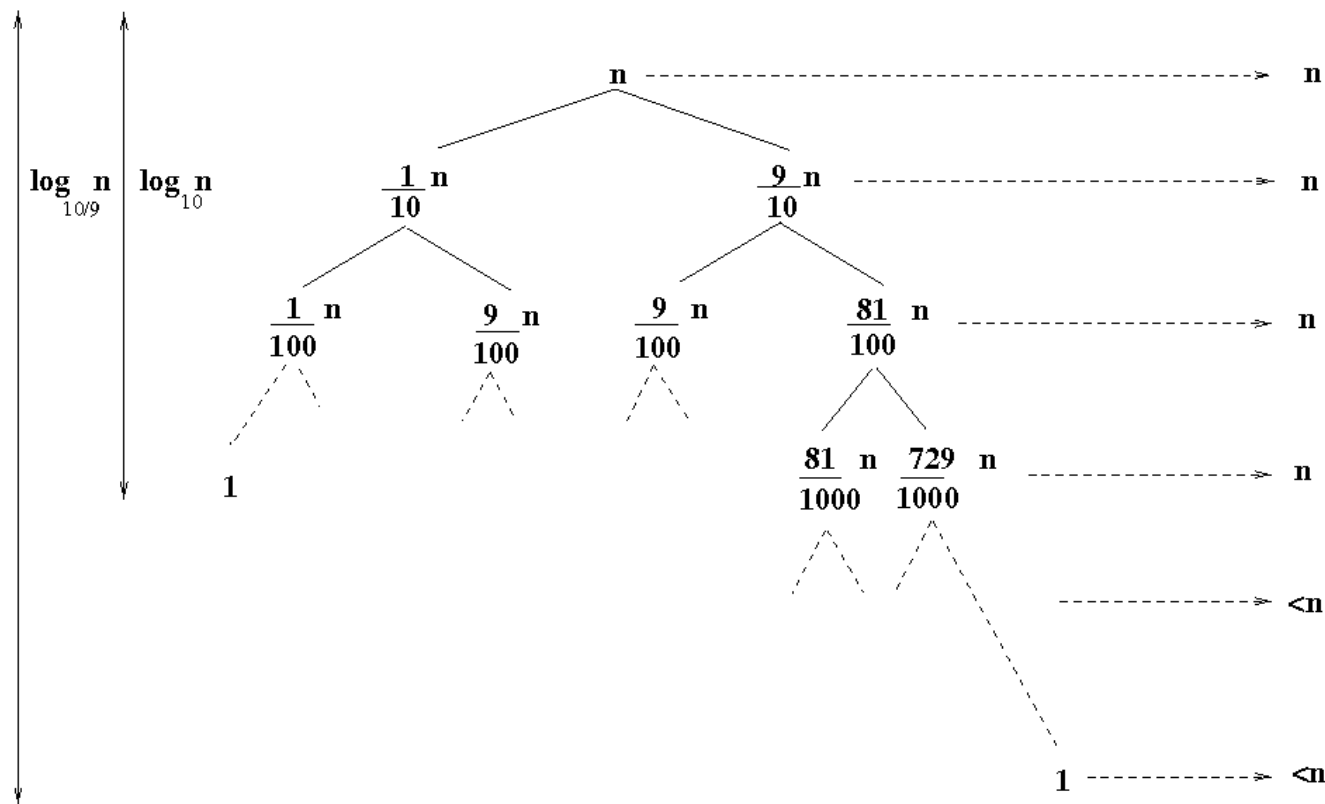


$$T(n) = 2T(n/2) + \Theta(n)$$

$$T(n) = \Theta(n \lg n)$$

A case between the best and worst case

- A case between the best and worst case (a $9n/10$ to $n/10$ splitting)



$$T(n) = T(9n/10) + T(n/10) + \Theta(n) = \Theta(n \lg n)$$

Average-Case Analysis of Quicksort

To do a precise average-case analysis of quicksort, we formulate a recurrence given the exact expected time $T(n)$:

$$T(n) = \sum_{q=1}^n \frac{1}{n} (T(q-1) + T(n-q)) + n - 1$$

Each possible pivot p is selected with equal probability. The number of **comparisons** needed to do the partition is $n - 1$.

$$T(n) = \sum_{q=1}^n \frac{1}{n} (T(q-1) + T(n-q)) + n - 1$$

$$T(n) = \frac{2}{n} \sum_{q=1}^n T(q-1) + n - 1$$

$$nT(n) = 2 \sum_{q=1}^n T(q-1) + n(n-1) \quad \text{multiply by } n$$

Cont'd

$$(n-1)T(n-1) = 2 \sum_{q=1}^{n-1} T(q-1) + (n-1)(n-2) \quad \text{apply to } n-1$$

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2(n-1)$$

rearranging the terms give us:

$$\begin{aligned} \frac{T(n)}{n+1} &= \frac{T(n-1)}{n} + \frac{2(n-1)}{n(n+1)} \\ &= \frac{T(n-1)}{n} + \frac{2}{n+1} - \frac{2}{n(n+1)} \\ &< \frac{T(n-1)}{n} + \frac{2}{n+1} \\ &< \frac{T(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\ &< \frac{T(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\ &< \frac{T(1)}{2} + \frac{2}{3} + \frac{2}{4} + \dots + \frac{2}{n} + \frac{2}{n+1} \end{aligned}$$

But the harmonic series $\sum_{k=1}^{n+1} \frac{1}{k} = \ln(n+1) + O(1)$

$$\text{so } T(n) < (n+1)(2 \ln(n+1) + O(1)) = O(n \lg n)$$