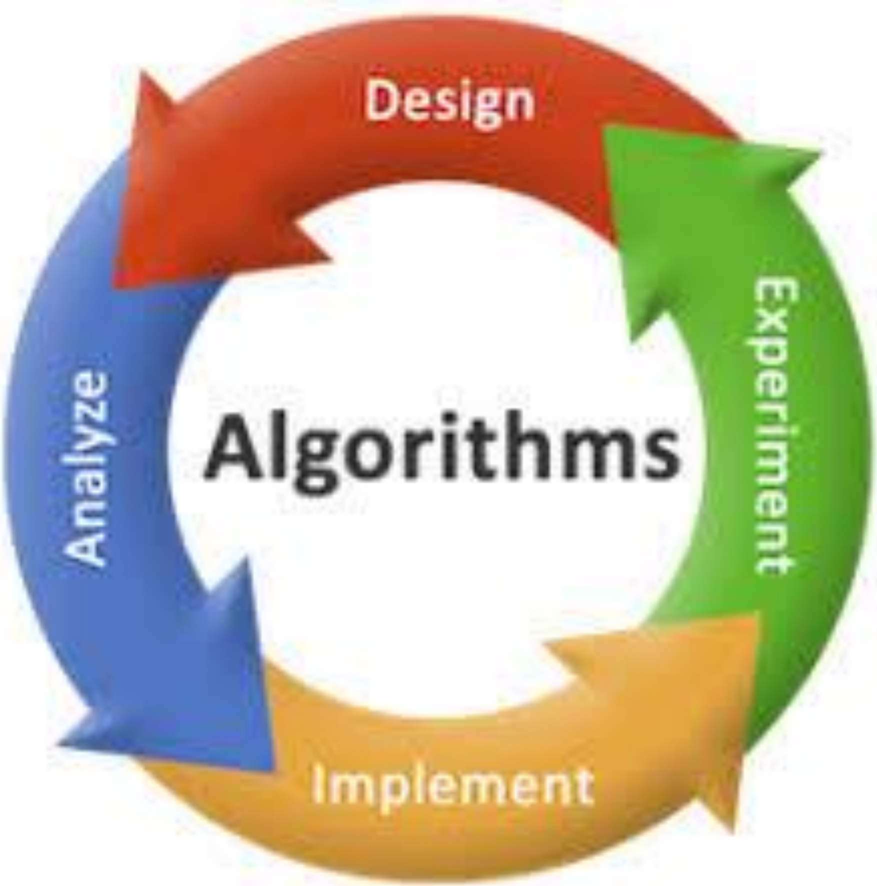


Design & Analysis of Algorithms

Lecture 17

Divide & Conquer

Quick Sort Algorithm

Divide & Conquer

```
Algorithm DAndC(P)
{
    if Small(P) then return S(P);
    else
    {
        divide P into smaller instances P_1, P_2, ..., P_k,  k ≥ 1;
        Apply DAndC to each of these subproblems;
        return Combine(DAndC(P_1),DAndC(P_2),...,DAndC(P_k));
    }
}
```

$$T(n) = \begin{cases} g(n) & n \text{ small} \\ T(n_1) + T(n_2) + \cdots + T(n_k) \quad + \quad f(n) & \text{otherwise} \end{cases}$$

Divide & Conquer

- Its complexity can be shown by recurrence relation of the form

$$T(n) = \begin{cases} T(1) & n = 1 \\ aT(n/b) + f(n) & n > 1 \end{cases}$$

where a and b are known constants. We assume that $T(1)$ is known and n is a power of b (i.e., $n = b^k$).

Quick Sort

- Divide-and-conquer process for sorting a typical subarray A[p…r] is
- **Divide**: Partition (rearrange) the array A[p…r] into two (possibly empty) subarrays A[p…q−1] and A[q+1…r] such that each element of A[p…q −1] is less than or equal to A[q], which is, in turn, less than or equal to each element of A[q+1…r]. **Compute** the index **q** as part of this partitioning procedure.

Quick Sort

- **Conquer**: Sort the two subarrays A[p…q−1] and A[q+1…r] by recursive calls to quicksort.

- **Combine**: Since the subarrays are sorted in place, no work is needed to combine them: the entire array A[p…r] is now sorted.

Quick Sort

ALGORITHM *Quicksort*$(A[l..r])$

//Sorts a subarray by quicksort
//Input: Subarray of array $A[0..n-1]$, defined by its
// left and right indices l and r
//Output: Subarray $A[l..r]$ sorted in nondecreasing order
if $l < r$
    $s \leftarrow Partition(A[l..r])$ //s is a split position
    $Quicksort(A[l..s-1])$
    $Quicksort(A[s+1..r])$

Book: A Levitin

Quick Sort

ALGORITHM *HoarePartition*$(A[l..r])$

//Partitions a subarray by Hoare's algorithm, using the first element
//   as a pivot
//Input: Subarray of array $A[0..n-1]$, defined by its left and right
//   indices l and r ($l < r$)
//Output: Partition of $A[l..r]$, with the split position returned as
//   this function's value

$p \leftarrow A[l]$
$i \leftarrow l$; $j \leftarrow r+1$
repeat
    repeat $i \leftarrow i+1$ **until** $A[i] \geq p$
    repeat $j \leftarrow j-1$ **until** $A[j] \leq p$
    swap$(A[i], A[j])$
until $i \geq j$
swap$(A[i], A[j])$ //undo last swap when $i \geq j$
swap$(A[l], A[j])$
return j

Book: A Levitin

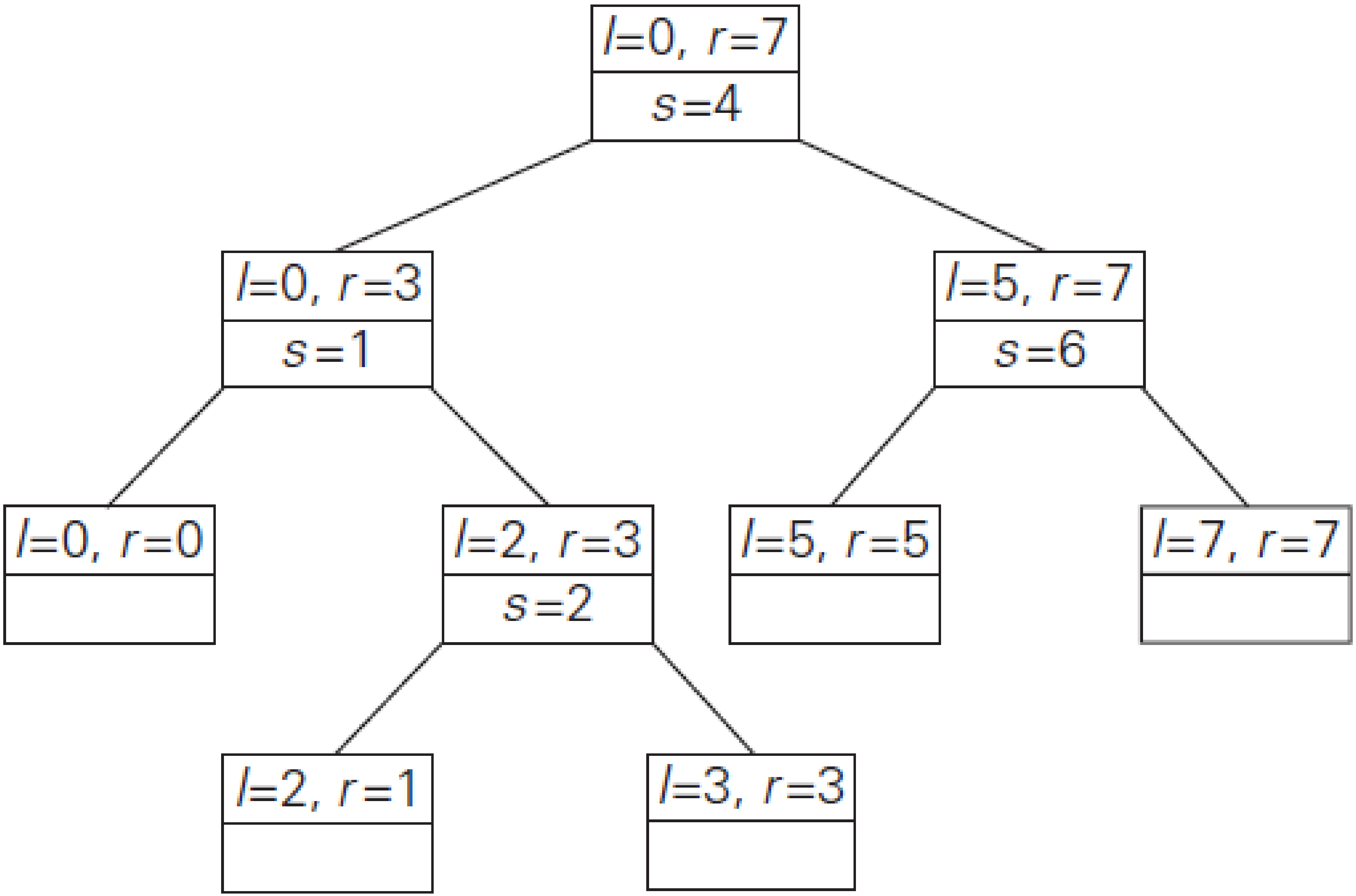
Quick Sort

0	1	2	3	4	5	6	7
5	*i* 3	1	9	8	2	4	*j* 7
5	3	1	*i* 9	8	2	*j* 4	7
5	3	1	*i* 4	8	2	*j* 9	7
5	3	1	4	*i* 8	*j* 2	9	7
5	3	1	4	*i* 2	*j* 8	9	7
5	3	1	4	*j* 2	*i* 8	9	7
2	3	1	4	**5**	8	9	7

Quick Sort

0	1	2	3	4	5	6	7
2	3	1	4	**5**	8	9	7
	i		*j*				
2	3	1	4				
	i	*j*					
2	3	1	4				
	i	*j*					
2	1	3	4				
	j	*i*				*i*	*j*
2	1	3	4		**8**	9	7
1	**2**	3	4			*i*	*j*
1					**8**	7	9
			i j			*j*	*i*
		3	4		**8**	7	9
		j	*i*		7	**8**	9
		3	4		7		
			4				9

Quick Sort



Quick Sort

```
QUICKSORT(A, p, r)
  if p < r
     then q ← PARTITION(A, p, r)
          QUICKSORT(A, p, q − 1)
          QUICKSORT(A, q + 1, r)
```

To sort an entire array A, the initial call is QUICKSORT(A, 1, $length[A]$).

Book: Cormen et al.

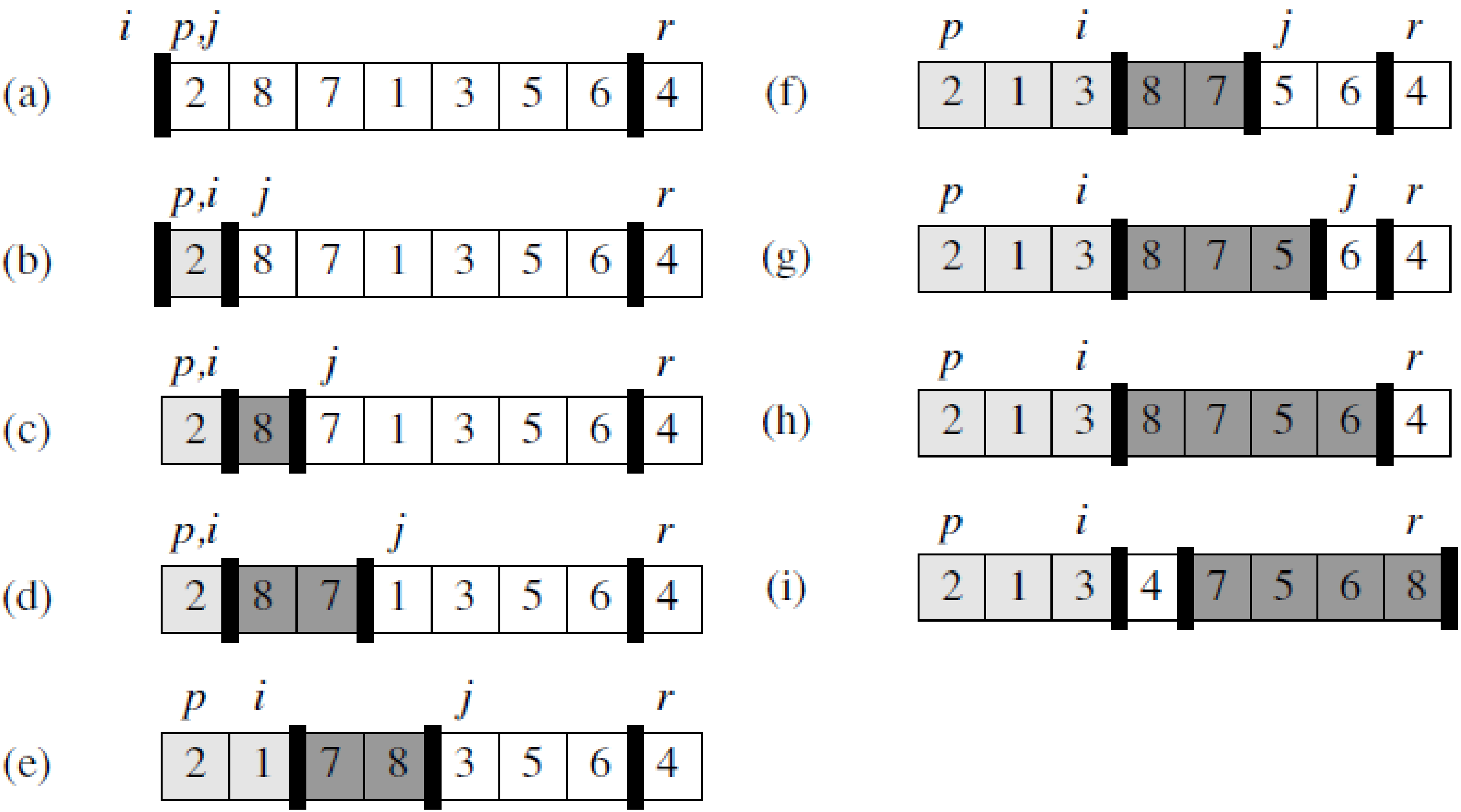
Quick Sort

The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p \,..\, r]$ in place.

```
PARTITION(A, p, r)
x ← A[r]
i ← p − 1
for j ← p to r − 1
      do if A[j] ≤ x
            then i ← i + 1
                  exchange A[i] ↔ A[j]
exchange A[i + 1] ↔ A[r]
return i + 1
```

Book: Cormen et al.

Quick Sort



Book: Cormen et al.

Quick Sort

```
Algorithm Partition(a, m, p)
// Within a[m], a[m+1], ..., a[p-1] the elements are
// rearranged in such a manner that if initially t = a[m],
// then after completion a[q] = t for some q between m
// and p-1, a[k] ≤ t for m ≤ k < q, and a[k] ≥ t
// for q < k < p. q is returned. Set a[p] = ∞.
{
    v := a[m]; i := m; j := p;
    repeat
    {
        repeat
            i := i + 1;
        until (a[i] ≥ v);

        repeat
            j := j - 1;
        until (a[j] ≤ v);

        if (i < j) then Interchange(a, i, j);

    } until (i ≥ j);

    a[m] := a[j]; a[j] := v; return j;
}
```

```
Algorithm Interchange(a, i, j)
// Exchange a[i] with a[j].
{
    p := a[i];
    a[i] := a[j]; a[j] := p;
}
```

Book: Horowitz & Sahni

Quick Sort

```
Algorithm QuickSort(p, q)
// Sorts the elements a[p], ..., a[q] which reside in the global
// array a[1 : n] into ascending order; a[n + 1] is considered to
// be defined and must be ≥ all the elements in a[1 : n].
{
    if (p < q) then  // If there are more than one element
    {
        // divide P into two subproblems.
            j := Partition(a, p, q + 1);
                // j is the position of the partitioning element
        // Solve the subproblems.
            QuickSort(p, j - 1);
            QuickSort(j + 1, q);
        // There is no need for combining solutions.
    }
}
```

Book: Horowitz & Sahni

Quick Sort

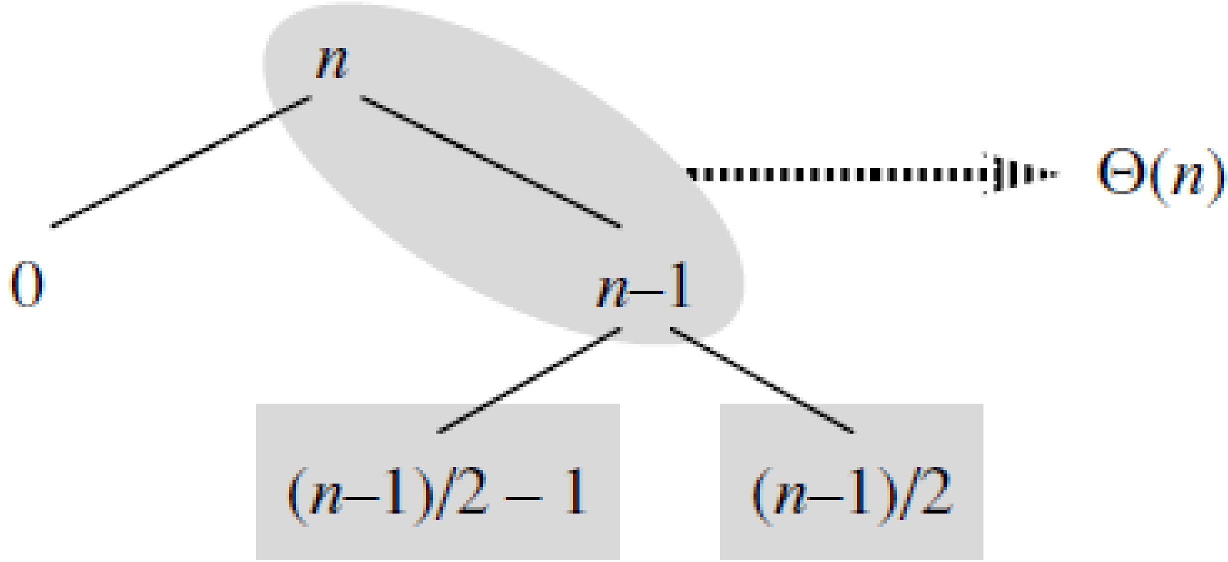
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	i	p
65	70	75	80	85	60	55	50	45	$+\infty$	2	9
65	45	75	80	85	60	55	50	70	$+\infty$	3	8
65	45	50	80	85	60	55	75	70	$+\infty$	4	7
65	45	50	55	85	60	80	75	70	$+\infty$	5	6
65	45	50	55	60	85	80	75	70	$+\infty$	6	5
60	45	50	55	65	85	80	75	70	$+\infty$		

Book: Horowitz & Sahni

Quick Sort

Time Complexity

- **Worst Case Partitioning**
  - when the partitioning routine produces one subproblem with n−1 elements and one with 0 elements.



Quick Sort

Time Complexity

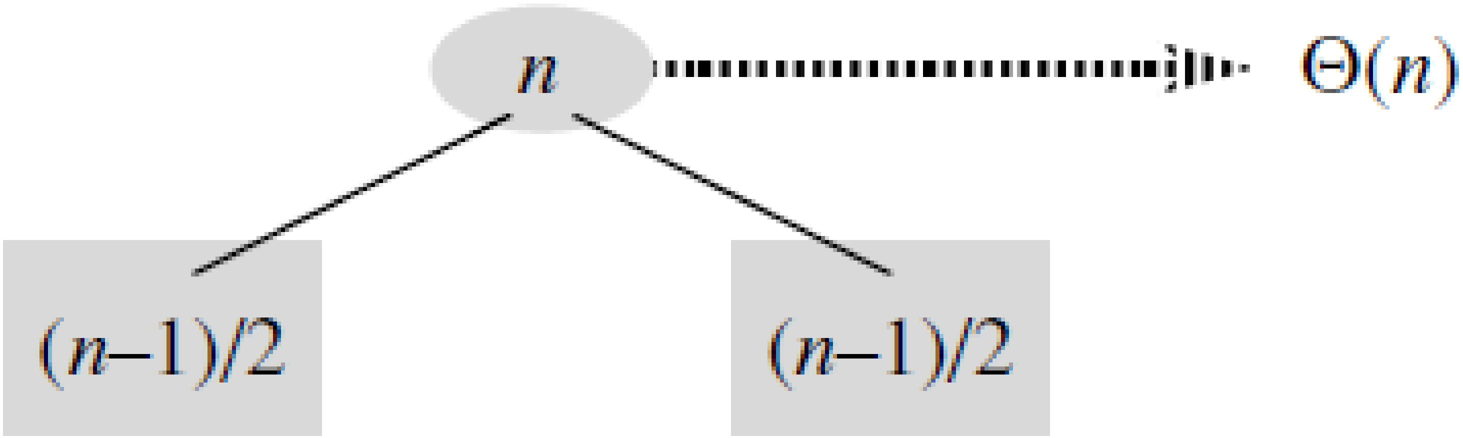
- **Worst Case Partitioning**
  - Since the recursive call on an array of size 0 just returns, T(0) = Θ(1), and the recurrence for the running time is

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) = \Theta(n^2) \end{aligned}$$

Quick Sort

Time Complexity

- **Best Case Partitioning**
  - In the most even possible split, PARTITION produces two subproblems, each of size no more than n/2, since one is of size $\lfloor n/2 \rfloor$ and one of size $\lceil n/2 \rceil - 1$.



Quick Sort

Time Complexity

- **Best Case Partitioning**
  - In this case, quicksort runs much faster.
  - The recurrence for the running time is

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

$$C_{best}(n) = 2C_{best}(n/2) + n \quad \text{for } n > 1, \quad C_{best}(1) = 0.$$

  - Solving it exactly for $n = 2^k$ we get

$$C_{best}(n) = n \log_2 n$$

Quick Sort

Time Complexity

- **Average Case Partitioning**
  - *We get **n*** different cases, PARTITION produces two subproblems, each of size from the set {(0,n-1),(1,n-2), …,(k-1,n-k),…,(n-2,1),(n-1,0)}.
  - From this we obtain the recurrence

$$C_A(n) = n + 1 + \frac{1}{n} \sum_{1 \le k \le n} [C_A(k-1)) + C_A(n-k)]$$

Note that $C_A(0) = C_A(1) = 0.$

Book: Horowitz & Sahni

Quick Sort

Time Complexity

- **Average Case Partitioning**

The number of element comparisons required by Partition on its first call is $n+1$.

Multiplying both sides by n, we obtain

$$nC_A(n) = n(n+1) + 2[C_A(0) + C_A(1) + \cdots + C_A(n-1)]$$

Replacing n by $n-1$ gives

$$(n-1)C_A(n-1) = n(n-1) + 2[C_A(0) + \cdots + C_A(n-2)]$$

Quick Sort

Time Complexity

- **Average Case Partitioning**

Subtracting we get

$$nC_A(n) - (n-1)C_A(n-1) = 2n + 2C_A(n-1)$$

or

$$C_A(n)/(n+1) = C_A(n-1)/n + 2/(n+1)$$

Repeatedly using this equation to substitute for $C_A(n-1), C_A(n-2), \ldots,$ we get

Book: Horowitz & Sahni

Quick Sort

Time Complexity

Book: Horowitz & Sahni

- **Average Case Partitioning**

$$
\begin{aligned}
\frac{C_A(n)}{n+1} &= \frac{C_A(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\
&= \frac{C_A(n-3)}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1} \\
&\vdots \\
&= \frac{C_A(1)}{2} + 2\sum_{3 \le k \le n+1} \frac{1}{k} \\
&= 2\sum_{3 \le k \le n+1} \frac{1}{k}
\end{aligned}
$$

$$
\text{Since} \sum_{3 \le k \le n+1} \frac{1}{k} \le \int_2^{n+1} \frac{1}{x}\, dx = \log_e(n+1) - \log_e 2
$$

$$
C_A(n) \le 2(n+1)[\log_e(n+2) - \log_e 2] = O(n \log n)
$$

References

- **Chapter 5**: Anany Levitin, "Introduction to the Design and Analysis of Algorithms", Pearson Education, Third Edition, 2017.
- **Chapter 3**: E Horowitz, S Sahni, S Rajasekaran, "Computer Algorithms", Computer Science Press, Third Edition, 2008.
- **Chapter 7**: Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, "Introduction to Algorithms", MIT Press/PHI Learning Private Limited, Third Edition, 2012.

Homework

- Design an algorithm to rearrange elements of a given array of n real numbers so that all its **negative** elements **precede** all its **positive** elements.

- Your algorithm should be both **time efficient** and **space efficient**.