Divide-and-Conquer

- In divide-and-conquer, a problem is solved **recursively**, applying three steps at each level of the recursion:

Divide-and-Conquer

- In divide-and-conquer, a problem is solved **recursively**, applying three steps at each level of the recursion:
  - **Divide** the problem into a number of subproblems that are smaller instances of the same problem.

Divide-and-Conquer

- In divide-and-conquer, a problem is solved **recursively**, applying three steps at each level of the recursion:
  - **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 subproblem sizes are small enough, just solve the subproblems in a straightforward manner.

Divide-and-Conquer

- In divide-and-conquer, a problem is solved **recursively**, applying three steps at each level of the recursion:
  - **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 subproblem sizes are small enough, just solve the subproblems in a straightforward manner.
  - **Combine** the solutions to the subproblems into the solution for the original problem.

Divide-and-Conquer

- When the subproblems are large enough to solve recursively, we call that the recursive case.

Divide-and-Conquer

- When the subproblems are large enough to solve recursively, we call that the recursive case.

- Once the subproblems become small enough that we no longer recurse, we say that the recursion “bottoms out” and that we have gotten down to the base case.

Divide-and-Conquer

- When the subproblems are large enough to solve recursively, we call that the recursive case.

- Once the subproblems become small enough that we no longer recurse, we say that the recursion “bottoms out” and that we have gotten down to the base case.

- Sometimes, in addition to subproblems that are smaller instances of the same problem, we have to solve subproblems that are not quite the same as the original problem.

Divide-and-Conquer

- When the subproblems are large enough to solve recursively, we call that the recursive case.

- Once the subproblems become small enough that we no longer recurse, we say that the recursion “bottoms out” and that we have gotten down to the base case.

- Sometimes, in addition to subproblems that are smaller instances of the same problem, we have to solve subproblems that are not quite the same as the original problem.

- We consider solving such subproblems as part of the combine step.

Divide-and-Conquer

- **The General Template**
  - Consider an arbitrary problem, and let *adhoc* be a simple algorithm capable of solving the problem.

Divide-and-Conquer

- **The General Template**
  - Consider an arbitrary problem, and let *adhoc* be a simple algorithm capable of solving the problem.

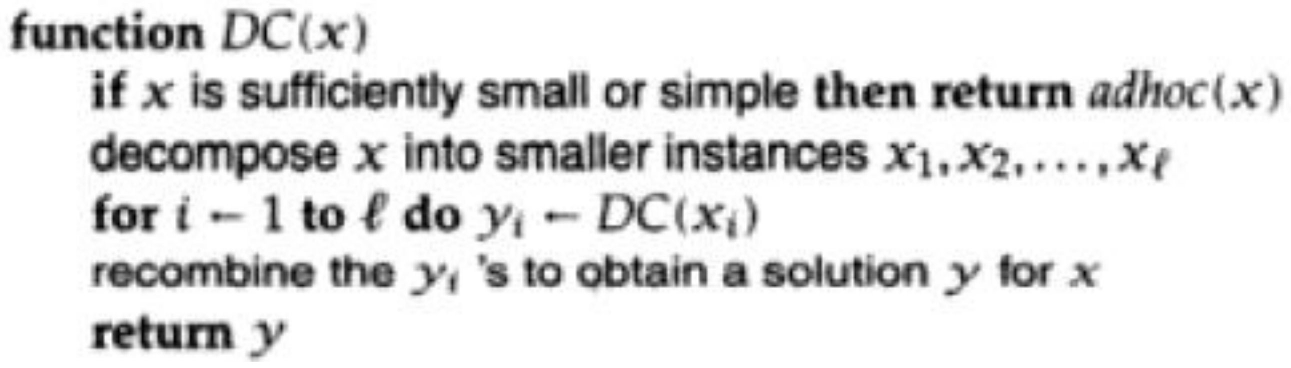
  - *adhoc* is also called the basic subalgorithm.

Divide-and-Conquer

- **The General Template**
  - Consider an arbitrary problem, and let *adhoc* be a simple algorithm capable of solving the problem.
  - *adhoc* is also called the basic subalgorithm.
  - *adhoc* is surely efficient on small instances of a problem but its performance on large instances is of no concern.

Divide-and-Conquer

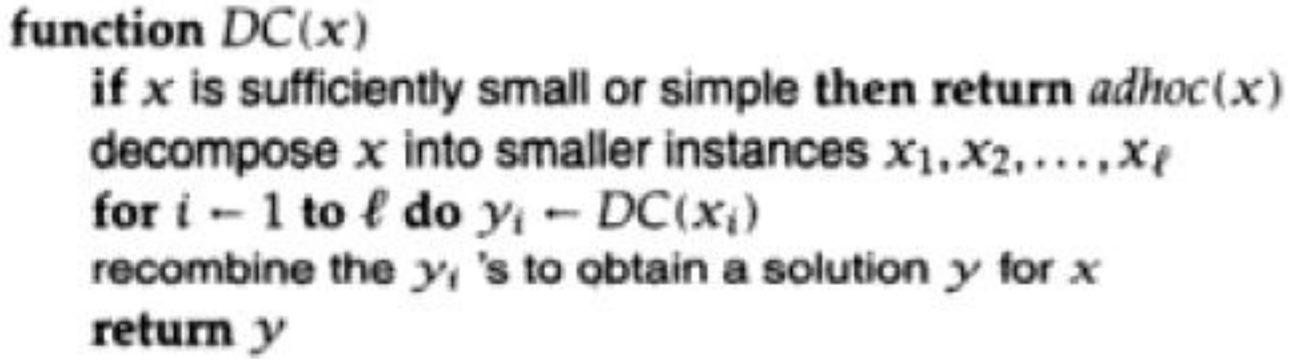
- The General Template
  - The general template for divide-and-conquer algorithms is as follows.



```
function $DC(x)$
    if $x$ is sufficiently small or simple then return $adhoc(x)$
    decompose $x$ into smaller instances $x_1, x_2, \ldots, x_\ell$
    for $i \leftarrow 1$ to $\ell$ do $y_i \leftarrow DC(x_i)$
    recombine the $y_i$ 's to obtain a solution $y$ for $x$
    return $y$
```

Divide-and-Conquer

- The General Template
  - The general template for divide-and-conquer algorithms is as follows.

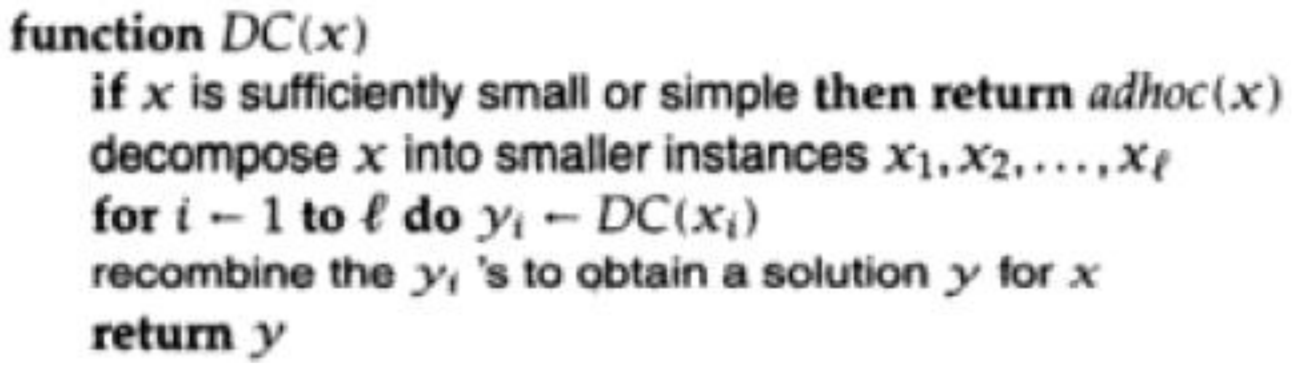


```
function $DC(x)$
    if $x$ is sufficiently small or simple then return $adhoc(x)$
    decompose $x$ into smaller instances $x_1, x_2, \ldots, x_\ell$
    for $i \leftarrow 1$ to $\ell$ do $y_i \leftarrow DC(x_i)$
    recombine the $y_i$ 's to obtain a solution $y$ for $x$
    return $y$
```

  - Some divide-and-conquer algorithms do not follow this outline exactly.

Divide-and-Conquer

- The General Template
  - The general template for divide-and-conquer algorithms is as follows.



```
function $DC(x)$
    if $x$ is sufficiently small or simple then return $adhoc(x)$
    decompose $x$ into smaller instances $x_1, x_2, \ldots, x_\ell$
    for $i \leftarrow 1$ to $\ell$ do $y_i \leftarrow DC(x_i)$
    recombine the $y_i$ 's to obtain a solution $y$ for $x$
    return $y$
```

  - Some divide-and-conquer algorithms do not follow this outline exactly.
  - For instance, they could require that the first subinstance be solved before the second subinstance is formulated.

Divide-and-Conquer

- The General Template
  - For divide-and-conquer to be worthwhile, three conditions must be met.

Divide-and-Conquer

- The General Template
  - For divide-and-conquer to be worthwhile, three conditions must be met.
    - The decision when to use the basic subalgorithm rather than to make recursive calls must be taken judiciously.

Divide-and-Conquer

- The General Template
  - For divide-and-conquer to be worthwhile, three conditions must be met.
    - The decision when to use the basic subalgorithm rather than to make recursive calls must be taken judiciously.
    - It must be possible to decompose an instance into subinstances and to recombine the subsolutions fairly efficiently, and

Divide-and-Conquer

- The General Template
  - For divide-and-conquer to be worthwhile, three conditions must be met.
    - The decision when to use the basic subalgorithm rather than to make recursive calls must be taken judiciously.
    - It must be possible to decompose an instance into subinstances and to recombine the subsolutions fairly efficiently, and
    - The subinstances should as far as possible be of about the same size.

Divide-and-Conquer

- The General Template
  - Most divide-and-conquer algorithms are such that the size of the “l” subinstances is roughly n/b for some constant b, where n is the size of the original instance.

Divide-and-Conquer

- The General Template
  - Most divide-and-conquer algorithms are such that the size of the “l” subinstances is roughly n/b for some constant b, where n is the size of the original instance.
  - The running-time analysis of such divide-and-conquer algorithms is almost automatic, thanks to Master method.

Divide-and-Conquer

- The General Template
  - Most divide-and-conquer algorithms are such that the size of the “l” subinstances is roughly n/b for some constant b, where n is the size of the original instance.
  - The running-time analysis of such divide-and-conquer algorithms is almost automatic, thanks to Master method.
  - How can one determine whether to divide the instance and make recursive calls, or whether the instance is so simple that it is better to invoke the basic subalgorithm directly?

Divide-and-Conquer

- The General Template
  - With most divide-and-conquer algorithms, this decision is based on a simple threshold, usually denoted n_0.

Divide-and-Conquer

- The General Template
  - With most divide-and-conquer algorithms, this decision is based on a simple threshold, usually denoted n_0.
  - The basic subalgorithm is used to solve any instance whose size does not exceed n_0.

Divide-and-Conquer

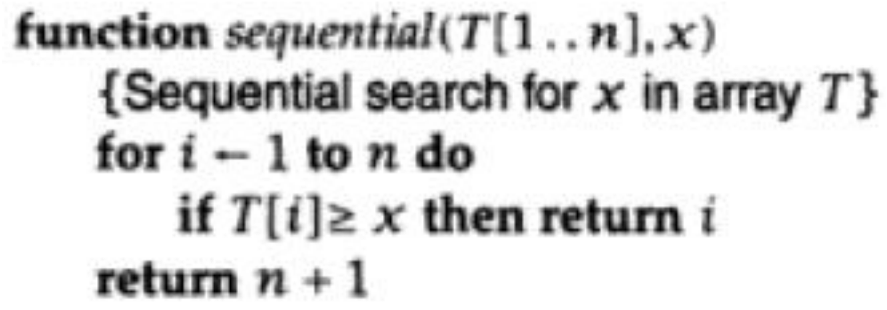
- The General Template
  - With most divide-and-conquer algorithms, this decision is based on a simple threshold, usually denoted n_0.
  - The basic subalgorithm is used to solve any instance whose size does not exceed n_0.
  - This threshold can be determined empirically. We can vary the value of the threshold and the size of the instances used for our tests and can note the time required for a number of cases.

Divide-and-Conquer

- The General Template
  - With most divide-and-conquer algorithms, this decision is based on a simple threshold, usually denoted n_0.
  - The basic subalgorithm is used to solve any instance whose size does not exceed n_0.
  - This threshold can be determined empirically. We can vary the value of the threshold and the size of the instances used for our tests and can note the time required for a number of cases.
  - It is often possible to estimate an optimal threshold by tabulating the results of these tests or by drawing a few diagrams.

Divide-and-Conquer

- Binary Search
  - Sequential Search



```
function sequential(T[1..n], x)
    {Sequential search for x in array T}
    for i ← 1 to n do
        if T[i] ≥ x then return i
    return n + 1
```

Divide-and-Conquer

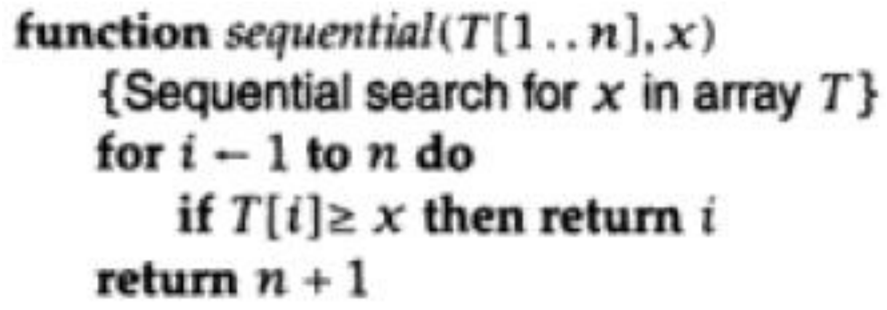
- Binary Search
  - Sequential Search

```
function sequential(T[1..n], x)
    {Sequential search for x in array T}
    for i ← 1 to n do
        if T[i] ≥ x then return i
    return n + 1
```

  - This algorithm clearly takes a time in $\theta(r)$, where r is the index returned.

Divide-and-Conquer

- Binary Search
  - Sequential Search



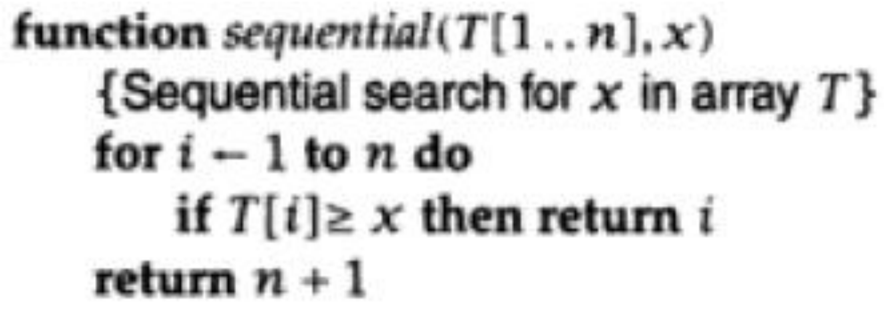
```
function sequential(T[1..n], x)
    {Sequential search for x in array T}
    for i ← 1 to n do
        if T[i] ≥ x then return i
    return n + 1
```

- This algorithm clearly takes a time in $\theta(r)$, where r is the index returned.

- This is $\Omega(n)$ in the worst case and O(1) in the best case.

Divide-and-Conquer

- Binary Search
  - Sequential Search



```
function sequential(T[1..n], x)
    {Sequential search for x in array T}
    for i ← 1 to n do
        if T[i] ≥ x then return i
    return n + 1
```

- This algorithm clearly takes a time in $\theta(r)$, where r is the index returned.

- This is $\Omega(n)$ in the worst case and O(1) in the best case.

- On the average, sequential search takes time in $\theta(n)$.

Divide-and-Conquer

- Binary Search
  - To speed up the search, we should look for x either in the first half of the array or in the second half.

Divide-and-Conquer

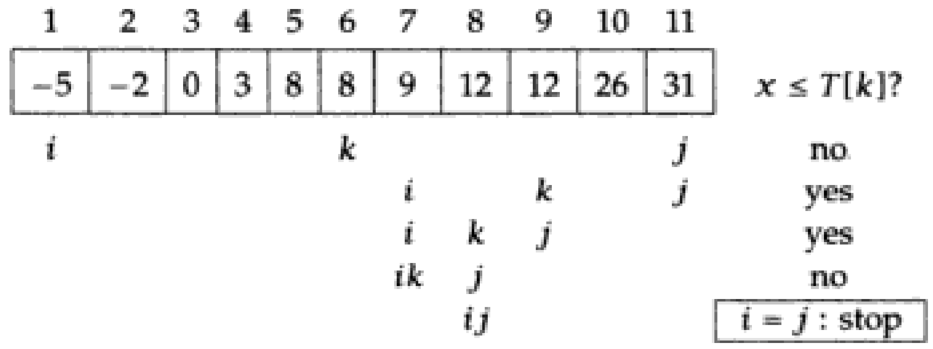
- Binary Search
  - To speed up the search, we should look for x either in the first half of the array or in the second half.
  - To find out which of these searches is appropriate, we compare x to an element in the middle of the array.

Divide-and-Conquer

- Binary Search
  - To speed up the search, we should look for x either in the first half of the array or in the second half.
  - To find out which of these searches is appropriate, we compare x to an element in the middle of the array.
  - We should keep continue this process until solution is found.

Divide-and-Conquer

- Binary Search
  - To speed up the search, we should look for x either in the first half of the array or in the second half.
  - To find out which of these searches is appropriate, we compare x to an element in the middle of the array.
  - We should keep continue this process until solution is found.



1	2	3	4	5	6	7	8	9	10	11	
−5	−2	0	3	8	8	9	12	12	26	31	$x \le T[k]$?
i					k					j	no
						i		k		j	yes
						i	k	j			yes
						ik	j				no
							ij				$i = j$: stop

Figure 7.1. Binary search for $x = 12$ in $T[1..11]$

```
function binsearch(T[1..n], x)
    if n = 0 or x > T[n] then return n + 1
    else return binrec(T[1..n], x)

function binrec(T[i..j], x)
    {Binary search for x in subarray T[i..j]
     with the promise that T[i − 1] < x ≤ T[j]}
    if i = j then return i
    k ← (i + j) ÷ 2
    if x ≤ T[k] then return binrec(T[i..k], x)
              else return binrec(T[k + 1..j], x)
```

Divide-and-Conquer

- Binary Search
  - Let t(m) be the time required for a call on binrec(T[i .. j], x), where m = j – i + 1 is the number of elements still under consideration in the search.

Divide-and-Conquer

- Binary Search
  - Let t(m) be the time required for a call on binrec(T[i .. j], x), where m = j – i + 1 is the number of elements still under consideration in the search.
  - The time required for a call on binsearch(T[1 .. n], x) is clearly t(n) up to a small additive constant.

Divide-and-Conquer

- Binary Search
  - Let t(m) be the time required for a call on binrec(T[i .. j], x), where m = j – i + 1 is the number of elements still under consideration in the search.
  - The time required for a call on binsearch(T[1 .. n], x) is clearly t(n) up to a small additive constant.
  - When m > 1, the algorithm takes a constant amount of time in addition to one recursive call on $\lceil m/2 \rceil$ or $\lfloor m/2 \rfloor$ elements, depending whether or not x $\leq$ T[k].

Divide-and-Conquer

- Binary Search
  - Let t(m) be the time required for a call on binrec(T[i .. j], x), where m = j – i + 1 is the number of elements still under consideration in the search.
  - The time required for a call on binsearch(T[1 .. n], x) is clearly t(n) up to a small additive constant.
  - When m > 1, the algorithm takes a constant amount of time in addition to one recursive call on $\lceil m/2 \rceil$ or $\lfloor m/2 \rfloor$ elements, depending whether or not x $\leq$ T[k].
  - Therefore, t(m) = t(m/2) + g(m), where g(m) ϵ O(1) = O(m^0).

Divide-and-Conquer

- Binary Search
  - Let t(m) be the time required for a call on binrec(T[i .. j], x), where m = j – i + 1 is the number of elements still under consideration in the search.
  - The time required for a call on binsearch(T[1 .. n], x) is clearly t(n) up to a small additive constant.
  - When m > 1, the algorithm takes a constant amount of time in addition to one recursive call on $\lceil m/2 \rceil$ or $\lfloor m/2 \rfloor$ elements, depending whether or not x ≤ T[k].
  - Therefore, t(m) = t(m/2) + g(m), where g(m) ϵ O(1) = O(m^0).
  - Using master method, we conclude that t(m) ϵ $\theta(\log m)$.

Divide-and-Conquer

- Quicksort
    - Quicksort applies the divide-and-conquer paradigm.

Divide-and-Conquer

- Quicksort
  - Quicksort applies the divide-and-conquer paradigm.
  - Here is the three-step divide-and-conquer process for sorting a typical subarray *A*[p .. r].

Divide-and-Conquer

- Quicksort
  - Quicksort applies the divide-and-conquer paradigm.

  - Here is the three-step divide-and-conquer process for sorting a typical subarray A[p .. r].

  - 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.

Divide-and-Conquer

- Quicksort
  - Quicksort applies the divide-and-conquer paradigm.
  - Here is the three-step divide-and-conquer process for sorting a typical subarray A[p .. r].
  - 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.
  - Conquer: Sort the two subarrays A[p .. q - 1] and A[q + 1 .. r] by recursive calls to quicksort.

Divide-and-Conquer

- Quicksort
  - Quicksort applies the divide-and-conquer paradigm.
  - Here is the three-step divide-and-conquer process for sorting a typical subarray A[p .. r].
  - 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.
  - Conquer: Sort the two subarrays A[p .. q - 1] and A[q + 1 .. r] by recursive calls to quicksort.
  - Combine: Because the subarrays are already sorted, no work is needed to combine them: the entire array A[p .. r] is now sorted.

Divide-and-Conquer

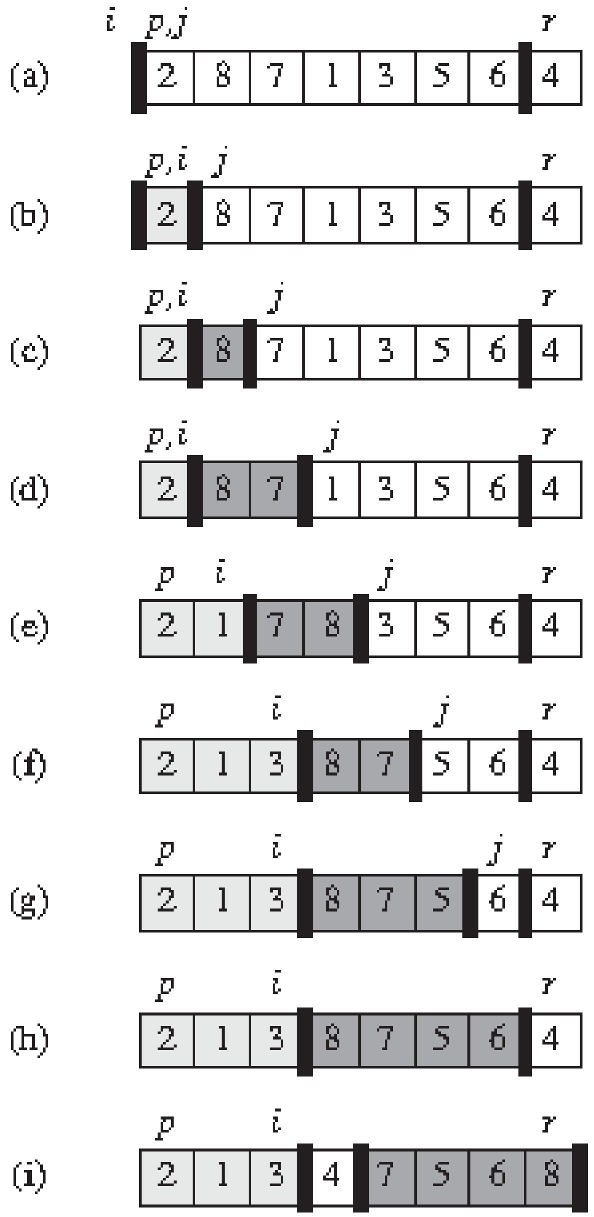
- Quicksort
  - The following procedure implements quicksort:

```
QUICKSORT($A, p, r$)
1  if $p < r$
2      $q$ = PARTITION($A, p, r$)
3      QUICKSORT($A, p, q - 1$)
4      QUICKSORT($A, q + 1, r$)
```

  - To sort an entire array *A*, the initial call is QUICKSORT(*A*, 1, *A*.length).

Divide-and-Conquer

➢ Quicksort



(a) i p,j r 2 8 7 1 3 5 6 4
(b) p,i j r 2 8 7 1 3 5 6 4
(c) p,i j r 2 8 7 1 3 5 6 4
(d) p,i j r 2 8 7 1 3 5 6 4
(e) p i j r 2 1 7 8 3 5 6 4
(f) p i j r 2 1 3 8 7 5 6 4
(g) p i j r 2 1 3 8 7 5 6 4
(h) p i r 2 1 3 8 7 5 6 4
(i) p i r 2 1 3 4 7 5 6 8

Figure 7.1 The operation of PARTITION on a sample array. Array entry $A[r]$ becomes the pivot element x. Lightly shaded array elements are all in the first partition with values no greater than x. Heavily shaded elements are in the second partition with values greater than x. The unshaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot x. **(a)** The initial array and variable settings. None of the elements have been placed in either of the first two partitions. **(b)** The value 2 is "swapped with itself" and put in the partition of smaller values. **(c)–(d)** The values 8 and 7 are added to the partition of larger values. **(e)** The values 1 and 8 are swapped, and the smaller partition grows. **(f)** The values 3 and 7 are swapped, and the smaller partition grows. **(g)–(h)** The larger partition grows to include 5 and 6, and the loop terminates. **(i)** In lines 7 8, the pivot element is swapped so that it lies between the two partitions.

Divide-and-Conquer

- Quicksort

Partitioning the array

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

```
PARTITION(A, p, r)
1  $x = A[r]$
2  $i = p - 1$
3  for $j = p$ to $r - 1$
4      if $A[j] \le x$
5          $i = i + 1$
6          exchange $A[i]$ with $A[j]$
7  exchange $A[i + 1]$ with $A[r]$
8  return $i + 1$
```

- p and r are the beginning and the end index of the array.
- Initially, A[r] is the pivot element (x)
- i is the index indicating end of the first sub-array.
- j is the iterative index.

Divide-and-Conquer

- Quicksort
  - The running time of PARTITION on the subarray A[p .. r] is θ(n), where n = r - p + 1
  - Worst Case Partitioning
    - The worst-case behavior for quicksort occurs when the partitioning routine produces one subproblem with n - 1 elements and one with 0 elements (e.g. when array is already sorted).
    - Let us assume that this unbalanced partitioning arises in each recursive call. The partitioning costs θ(n) time.
    - Since the recursive call on an array of size 0 just returns, T(0) = $\theta(1)$, and the recurrence for the running time is
$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) . \end{aligned}$$
    - Solving this inhomogeneous recurrence gives, T(n) = $\theta(n^2)$

Divide-and-Conquer

- Quicksort
  - 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 $\lceil n/2 \rceil - 1$.
    - In this case, quicksort runs much faster.
    - The recurrence (tolerating the sloppiness from ignoring the floor and ceiling and from subtracting 1) for the running time is then

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

    - Using master method to solve this recurrence, T(n) = θ(n lg n)

Divide-and-Conquer

- Finding the Median
  - The naïve algorithm for determining the median of T[1 .. n] consists of sorting the array and then extracting its $\lceil n/2 \rceil - \mathrm{th}$ entry. If we use heapsort or mergesort, this takes a time in θ(n log n).

Divide-and-Conquer

- Finding the Median
  - The naïve algorithm for determining the median of T[1 .. n] consists of sorting the array and then extracting its $\lceil n/2 \rceil - \mathrm{th}$ entry. If we use heapsort or mergesort, this takes a time in θ(n log n).

  - Can we do better?

Divide-and-Conquer

- Finding the Median
  - The naïve algorithm for determining the median of T[1 .. n] consists of sorting the array and then extracting its $\lceil n/2 \rceil - \mathrm{th}$ entry. If we use heapsort or mergesort, this takes a time in θ(n log n).
  - Can we do better?
  - To answer this question, we will study the interrelation between finding the median and selecting the s-th smallest element.

Divide-and-Conquer

- Finding the Median
    - It is obvious that any algorithm for the selection problem can be used to find the median: simply select the $\lceil n/2 \rceil -$ th smallest element of T.

Divide-and-Conquer

- Finding the Median
    - It is obvious that any algorithm for the selection problem can be used to find the median: simply select the $\lceil n/2 \rceil -$ th smallest element of T.
    - Interestingly, the converse holds as well.

Divide-and-Conquer

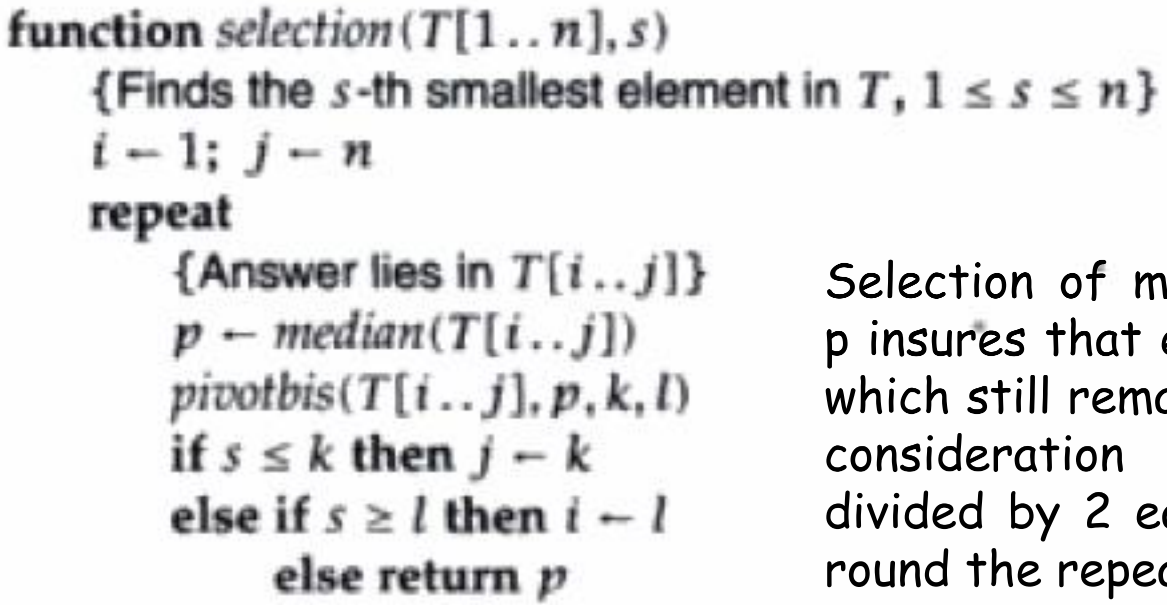
- Finding the Median
  - It is obvious that any algorithm for the selection problem can be used to find the median: simply select the $\lceil n/2 \rceil -$ th smallest element of T.
  - Interestingly, the converse holds as well.
  - Assume for now the availability of an algorithm median(T[1 .. n]) that returns the median of T.

Divide-and-Conquer

- Finding the Median
  - It is obvious that any algorithm for the selection problem can be used to find the median: simply select the $\lceil n/2 \rceil -$ th smallest element of T.
  - Interestingly, the converse holds as well.
  - Assume for now the availability of an algorithm median(T[1 .. n]) that returns the median of T.
  - Given an array T and an integer s, how could this algorithm be used to determine the s-th smallest element of T?

Divide-and-Conquer

- Finding the Median
  - **procedure** *pivotbis*($T[i..j], p$; **var** k, l)
  - This procedure partitions T into three sections using p as pivot.
  - After pivoting, the elements in T[i .. k] are smaller than p, those in T[k + 1 .. l - 1] are equal to p, and those in T[l .. j] are larger than p.
  - Note that k and l are found as a part of the procedure.

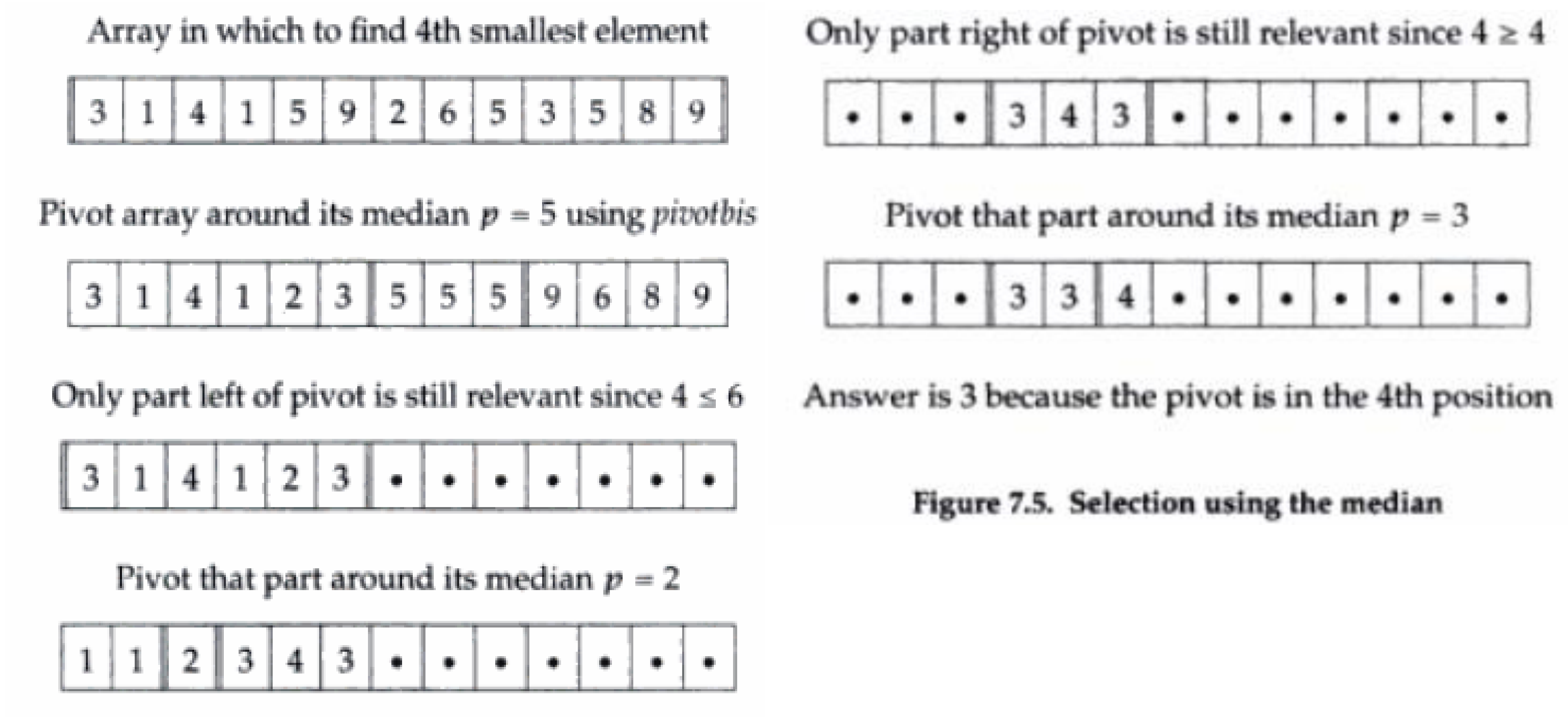


```
function selection(T[1..n], s)
    {Finds the s-th smallest element in T, 1 ≤ s ≤ n}
    i ← 1; j ← n
    repeat
        {Answer lies in T[i..j]}
        p ← median(T[i..j])
        pivotbis(T[i..j], p, k, l)
        if s ≤ k then j ← k
        else if s ≥ l then i ← l
            else return p
```

Selection of median as p insures that elements which still remain under consideration are divided by 2 each time round the repeat loop.

Divide-and-Conquer

- Finding the Median



Array in which to find 4th smallest element

3	1	4	1	5	9	2	6	5	3	5	8	9

Pivot array around its median $p = 5$ using *pivotbis*

3	1	4	1	2	3	5	5	5	9	6	8	9

Only part left of pivot is still relevant since $4 \le 6$

3	1	4	1	2	3	•	•	•	•	•	•	•

Pivot that part around its median $p = 2$

1	1	2	3	4	3	•	•	•	•	•	•	•

Only part right of pivot is still relevant since $4 \ge 4$

•	•	•	3	4	3	•	•	•	•	•	•	•

Pivot that part around its median $p = 3$

•	•	•	3	3	4	•	•	•	•	•	•	•

Answer is 3 because the pivot is in the 4th position

Figure 7.5. Selection using the median

Divide-and-Conquer

- Finding the Median
  - The above algorithm selects the required element of T after going round the repeat loop a logarithmic number of times in the worst case. However, trips round the loop no longer takes constant time but it takes linear time (because of pivotbis).
  - This algorithm can not be used until we have an efficient way to find the median, which was our original problem. Can we modify the algorithm to avoid the need to the median?
  - It is to note that presented algorithm still works regardless of which element of T is chosen as pivot (the value of p).
  - It is only the efficiency of the algorithm that depends on the choice of pivot: using the median assures us that the number of elements still under consideration is at least divided by 2 each time round the repeat loop.

Divide-and-Conquer

- Finding the Median
  - If we are willing to sacrifice speed in the worst case to obtain an algorithm reasonably fast on the average, we can simply choose T[i] as pivot.
  - In other words, replace the first instruction in the loop, with p <- T[i].

Divide-and-Conquer

- Finding the Median
  - This causes the algorithm to spend quadratic time in the worst case, for example if the array is in decreasing order and we wish to find the smallest element.
  - Note that in worst case, elements that are still left under consideration is reduced just by 1 each time round the loop.
  - Nevertheless, this modified algorithm runs in linear time on the average, under our usual assumption that the elements of T are distinct and that each of the n! possible initial permutations of the elements is equally likely.

Divide-and-Conquer

- Finding the Median
  - This is much better than the time required on the average if we proceed by sorting the array, but the worst case behaviour is unacceptable for many applications.
  - Happily, this quadratic worst case can be avoided without sacrificing linear behaviour on the average. The idea is that the number of trips round the loop remains logarithmic provided the pivot is chosen reasonably close to the median.
  - A good approximation to the median can be found quickly with a little cunning.

Divide-and-Conquer

- Finding the Median

  -
    ```
    function pseudomed(T[1..n])
        {Finds an approximation to the median of array T}
        if $n \le 5$ then return adhocmed(T)
        $z \leftarrow \lfloor n/5 \rfloor$
        array Z[1..z]
        for $i \leftarrow 1$ to $z$ do $Z[i] \leftarrow$ adhocmed(T[5i − 4..5i])
        return selection(Z, $\lceil z/2 \rceil$)
    ```

    Z[1] will have median of first 5 elements, Z[2] will have median of next 5 elements and so on.

  - Here, *adhocmed* is an algorithm specially designed to find the median of at most five elements, which can be done in a time bounded above by a constant, and *selection*$(Z, \lceil z/2 \rceil)$ determines the exact median of array Z. Let p be the value returned by a call on *pseudomed*(T). How far from the true median of T can p be when $n > 5$?

  - Although p is probably not the exact median of T, it can be concluded that its rank is approximately between 3n/10 and 7n/10.

Divide-and-Conquer

- Finding the Median
  - If we use pseudomed in selection it will take linear time.
  -

Divide-and-Conquer

- Exponentiation
  - Let a and n be two integers. We wish to compute the exponentiation.
  - For simplicity, we shall assume throughout this section that n > 0.
  - If n is small, the obvious algorithm is adequate.

```
function exposeq(a, n)
    r ← a
    for i ← 1 to n − 1 do r ← a × r
    return r
```

  - The algorithm takes a time in θ(n) since the instruction r <- a x r is executed exactly n – 1 times, provided that multiplications are counted as elementary operations. However, on most computers, even small values of n and a cause this algorithm to produce integer overflow.

Divide-and-Conquer

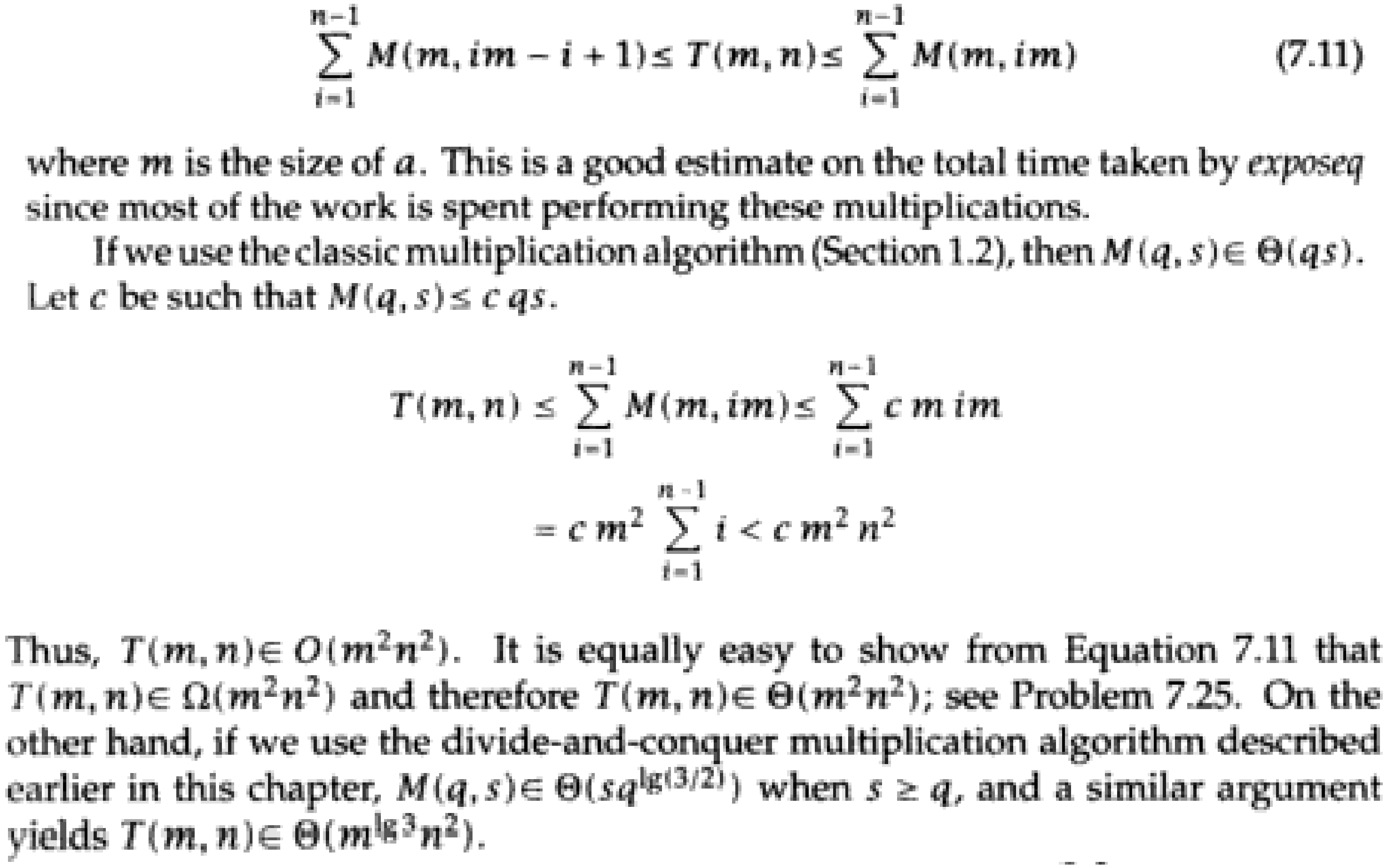
- Exponentiation
  - If we wish to handle larger operands, we must take account of the time required for each multiplication.
  - Let M(q, s) denote the time needed to multiply two integers of sizes q and s.
  - Size of integers may be considered, in decimal digits, in bits, or in any other fixed basis larger than 1.
  - Assume for simplicity that q1 <= q2 and s1 <= s2 imply that M(q1, s1) <= M(q2, s2).
  - Let us examine how much time our algorithm spends multiplying integers when exposeq(a, n) is called.

Divide-and-Conquer

- Exponentiation
  - Let m be the size of a. First note that the product of two integers of size i and j is of size at least $i + j - 1$ and at most $i + j$.
  - Let r_i and m_i be the value and the size of r at the beginning of the ith time round the loop. Clearly $r_1 = a$ and therefore $m_1 = m$.
  - Since $r_{i+1} = ar_i$, the size of r_{i+1} is at least $m + m_i - 1$ and at most $m + m_i$.
  - Using mathematical induction, it can be proved that $im - i + 1 <= m_i <= im$ for all i.
  - Therefore the multiplication performed the i-th time round the loop concerns an integer of size m and an integer whose size is between $im - i + 1$ and im, which takes a time between $M(m, im - i + 1)$ and $M(m, im)$.

Divide-and-Conquer

- Exponentiation
  - The total time T(m, n) spent multiplying when computing a^n with exposeq is therefore



$$\sum_{i=1}^{n-1} M(m, im - i + 1) \le T(m,n) \le \sum_{i=1}^{n-1} M(m, im) \qquad (7.11)$$

where m is the size of a. This is a good estimate on the total time taken by *exposeq* since most of the work is spent performing these multiplications.

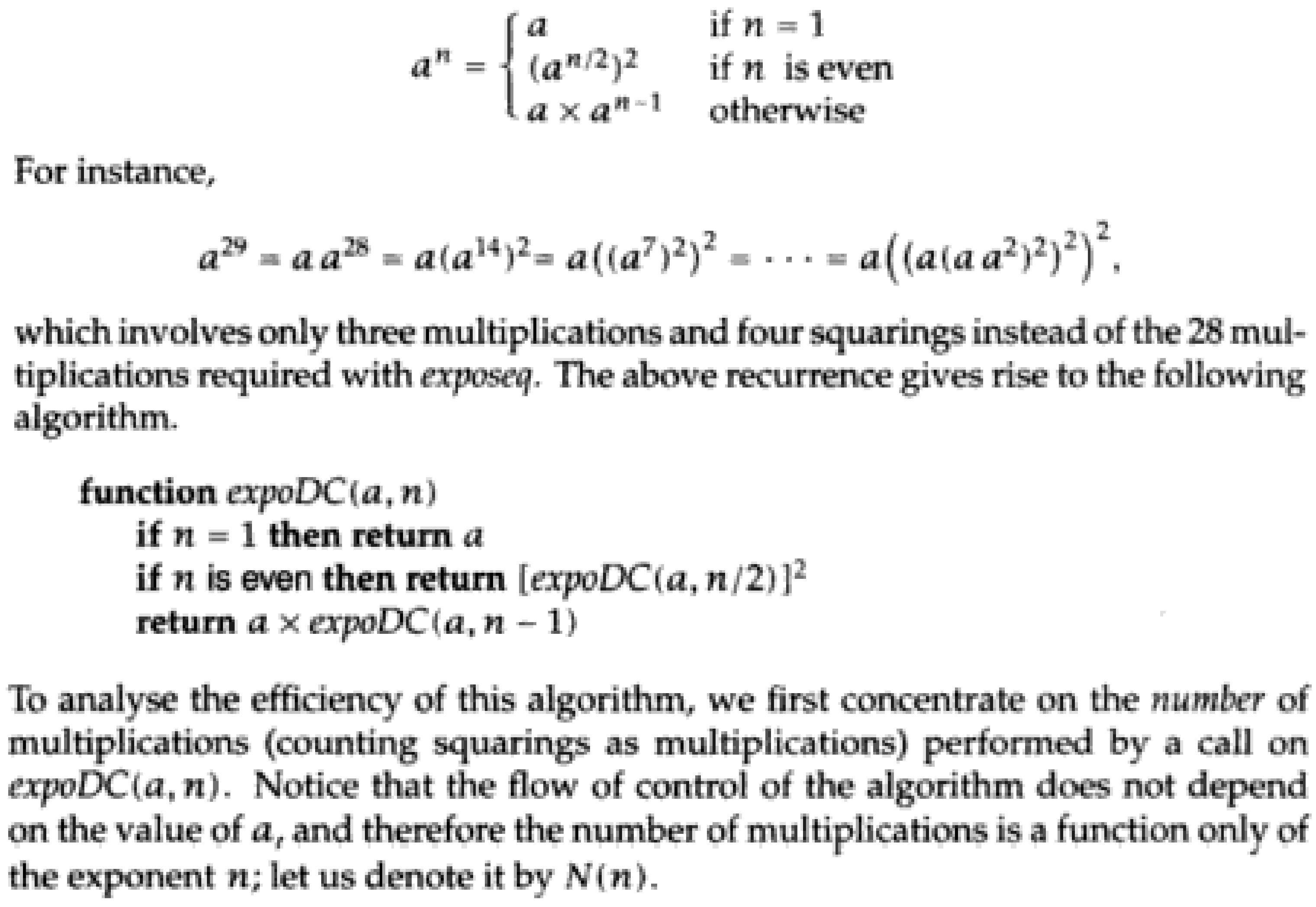
If we use the classic multiplication algorithm (Section 1.2), then $M(q,s) \in \Theta(qs)$. Let c be such that $M(q,s) \le c\,qs$.

$$T(m,n) \le \sum_{i=1}^{n-1} M(m, im) \le \sum_{i=1}^{n-1} c\,m\,im$$
$$= c\,m^2 \sum_{i=1}^{n-1} i < c\,m^2 n^2$$

Thus, $T(m,n) \in O(m^2 n^2)$. It is equally easy to show from Equation 7.11 that $T(m,n) \in \Omega(m^2 n^2)$ and therefore $T(m,n) \in \Theta(m^2 n^2)$; see Problem 7.25. On the other hand, if we use the divide-and-conquer multiplication algorithm described earlier in this chapter, $M(q,s) \in \Theta(sq^{\lg(3/2)})$ when $s \ge q$, and a similar argument yields $T(m,n) \in \Theta(m^{\lg 3} n^2)$.

Divide-and-Conquer

- Exponentiation
  - The idea of expoDC can be derived as under.



$$a^n = \begin{cases} a & \text{if } n = 1 \\ (a^{n/2})^2 & \text{if } n \text{ is even} \\ a \times a^{n-1} & \text{otherwise} \end{cases}$$

For instance,

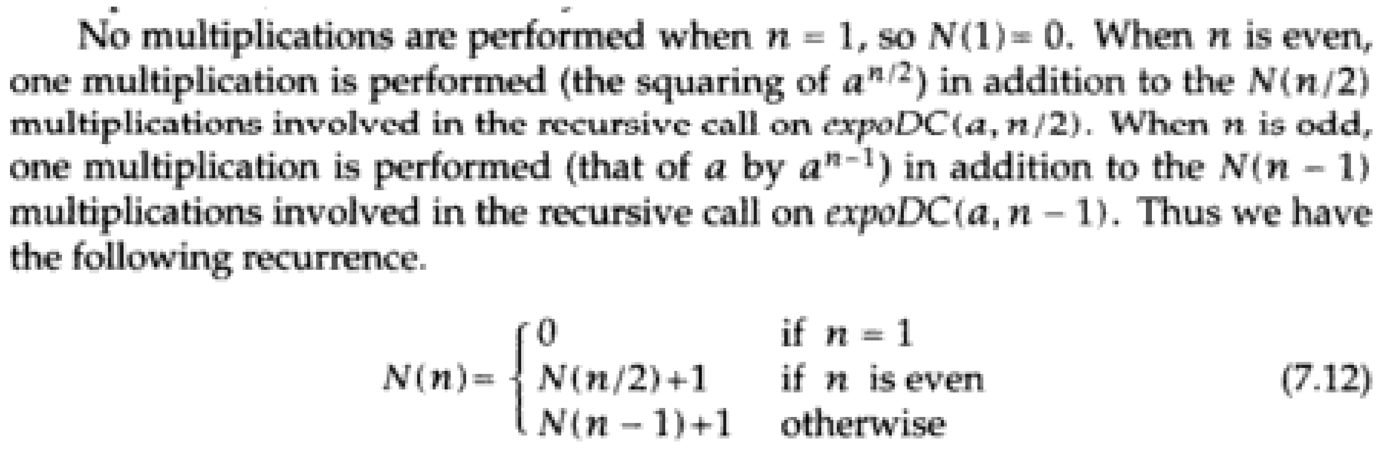
$$a^{29} = a\,a^{28} = a(a^{14})^2 = a((a^7)^2)^2 = \cdots = a\Big((a(a\,a^2)^2)^2\Big)^2,$$

which involves only three multiplications and four squarings instead of the 28 multiplications required with *exposeq*. The above recurrence gives rise to the following algorithm.

```
function expoDC(a, n)
    if n = 1 then return a
    if n is even then return [expoDC(a, n/2)]^2
    return a × expoDC(a, n − 1)
```

To analyse the efficiency of this algorithm, we first concentrate on the *number* of multiplications (counting squarings as multiplications) performed by a call on *expoDC*(a, n). Notice that the flow of control of the algorithm does not depend on the value of a, and therefore the number of multiplications is a function only of the exponent n; let us denote it by $N(n)$.

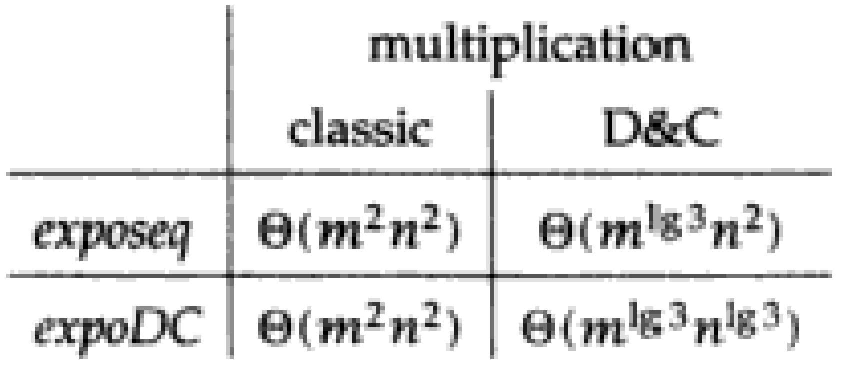
Divide-and-Conquer

- Exponentiation
  - 

    No multiplications are performed when $n = 1$, so $N(1) = 0$. When n is even, one multiplication is performed (the squaring of $a^{n/2}$) in addition to the $N(n/2)$ multiplications involved in the recursive call on *expoDC*$(a, n/2)$. When n is odd, one multiplication is performed (that of a by a^{n-1}) in addition to the $N(n-1)$ multiplications involved in the recursive call on *expoDC*$(a, n-1)$. Thus we have the following recurrence.

    $$N(n) = \begin{cases} 0 & \text{if } n = 1 \\ N(n/2)+1 & \text{if } n \text{ is even} \\ N(n-1)+1 & \text{otherwise} \end{cases} \tag{7.12}$$

  - Solving this, it can be shown that N(n) ∈ θ(log n). Recall that exposeq required multiplications which were in $\theta(n)$ to perform the same exponentiation.



	multiplication	
	classic	D&C
exposeq	$\Theta(m^2 n^2)$	$\Theta(m^{\lg 3} n^2)$
expoDC	$\Theta(m^2 n^2)$	$\Theta(m^{\lg 3} n^{\lg 3})$

Disclaimer

- The presentation is not original and its is prepared from various sources for teaching purpose only.