Chapter 2

Divide-and-Conquer

روش تقسیم و (حل)غلبه

روش تقسیم و (حل)غلبه

- روشی بالا به پایین
- Top-down Approach
- یک مسئله به صورت بازگشتی آن‌قدر به دو یا چند مساله کوچکتر (از همان جنس) تقسیم می‌شود تا زمانی که راه حل برای مسائل کوچکتر قابل دستیابی باشد.
- سپس راه‌حل‌های کوچکتر برای رسیدن به حل نهایی، در صورت لزوم باهم ترکیب می‌شوند.

An example

- Suppose $x = 18$ and we have the following array:

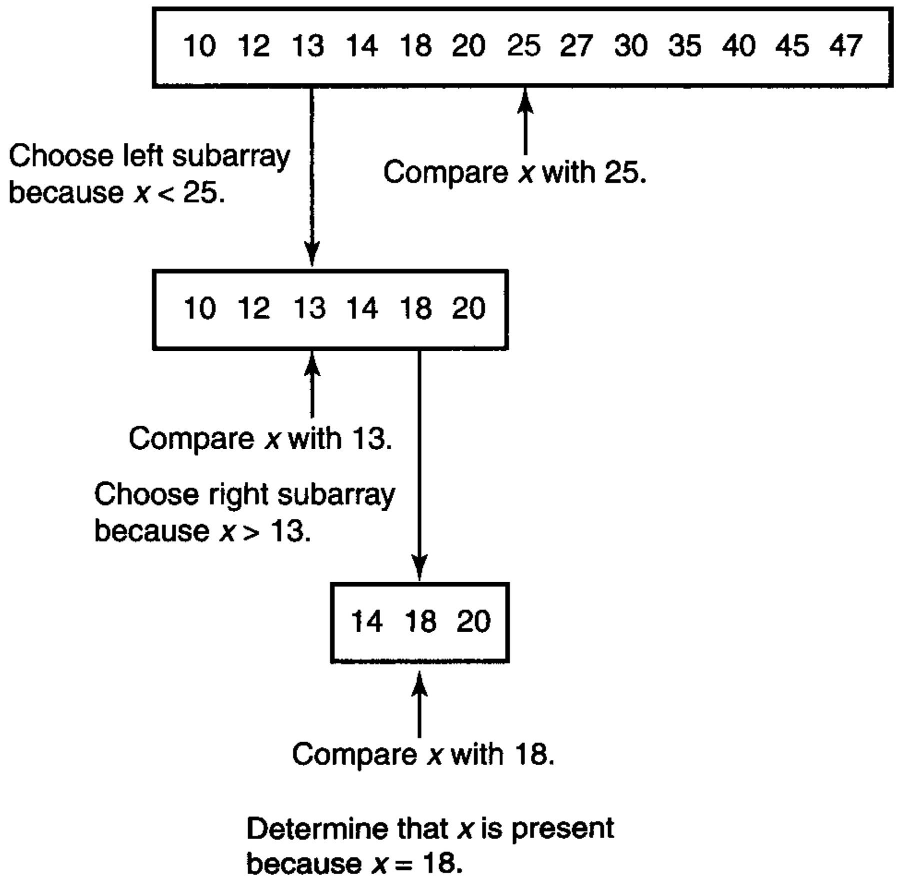
10 12 13 14 18 20 25 27 30 35 40 45 47

↑

middle item

What next? (Figure 2.1)

The whole process



Binary search

- Algorithm 2.1: Binary Search (Recursive)
  - Problem: Determine whether *x* is in the sorted array *S* of size *n.*
  - Inputs: positive integer *n*, sorted (nondecreasing order) array of keys *S* indexed from 1 to *n*, a key *x.*
  - Outputs: *location*, the location of *x* in *S* (0 if *x* is not in *S*).

```
index location (index low, index high)
{
index mid;
if (low > high)
        return 0;
else {
        mid = [(low + high)/2];
        if (x == S[mid])
                return mid
        else if (x <= S[mid])
                return location(low, mid - 1);
        else return location(mid + 1, high);
        }
}
```

Worst-case time complexity

- Basic operation: the comparison of *x* with *S*[*mid*]
- Input size: *n*, the number of items in the array
- Time complexity:

$$\begin{cases} W(n) = W(\frac{n}{2}) + 1 & for \quad n > 1, \quad n \quad a \quad power \quad of \quad 2 \\ W(1) = 1 \end{cases}$$

- Solution: $W(n) = \lg n + 1$
- If *n* is not a power of 2: $W(n) = \lfloor \lg n \rfloor + 1 \in \theta(\lg n)$

Merge Sort
One example

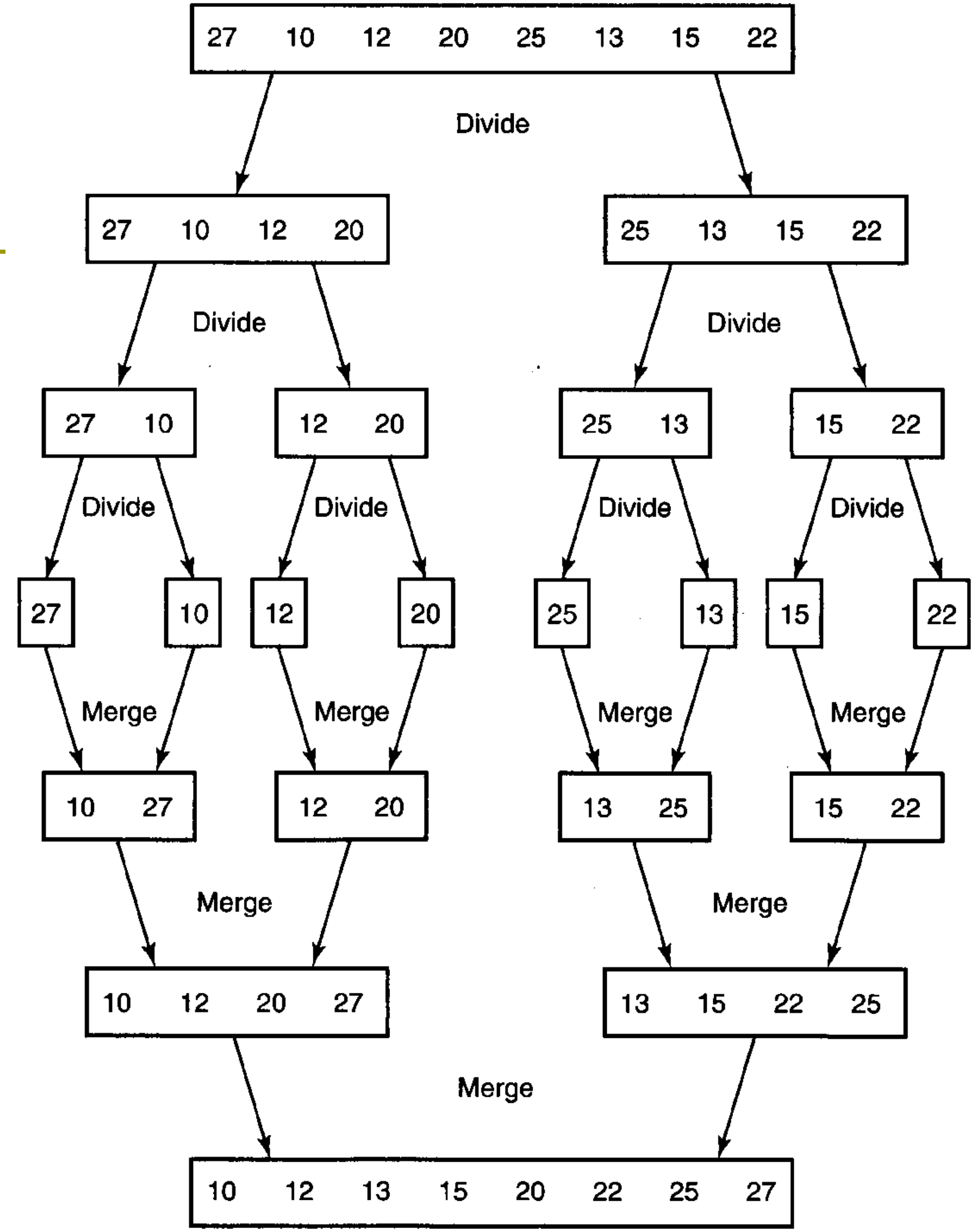


Figure 2.2 ● The steps done by a human when sorting with Mergesort.

The algorithm

- Algorithm 2.2: Mergesort
  - Problem: Sort *n* keys in nondecreasing sequence.
  - Inputs: positive integer *n*, array of keys *S* indexed from 1 to *n.*
  - Outputs: the array *S* containing the keys in nondecreasing order.

```
void mergesort (int n, keytype S[])
{
if (n>1) {
      const int h=[n/2], m = n - h;
      keytype U[1 ..h], V[1 ..m];
      copy S[1] through S[h] to U[1] through U[h];
      copy S[h+1] through S[n] to V[1] through V[m];
      mergesort(h, U);
      mergesort(m, V);
      merge (h, m, U, V, S);
      }
}
```

Merge

- Algorithm 2.3: Merge
  - Problem: Merge two sorted arrays into one sorted array.
  - Inputs: positive integers *h* and *m*, array of sorted keys *U* indexed from 1 to *h*, array of sorted keys *V* indexed from 1 to *m.*
  - Outputs: an array *S* indexed from 1 to *h* + *m* containing the keys in *U* and *V* in a single sorted array.

```
void merge (int h, int m, const keytype U[], const keytype V[], keytype S[])
{
        index i, j, k;
        i = 1; j = 1; k = 1;
        while (i <= h && j <= m)
        {
                if (U[i] < V[j]) {
                        S[k] = U[i]; i++; }
                else {
                        S[k] = V[j]; j++; }
                k++;
        }
        if (i>h)
                copy V[j] through V[m] to S[k] through S[h+m];
        else
                copy U[i] through U[h] to S[k] through S[h+m];
}
```

Worst-case time complexity (Merge)

- Basic operation: the comparison of $U[i]$ with $V[j]$.
- Input size: h and m, the number of items in each of the two input arrays.
- Time complexity:

  $W(h, m) = h + m - 1 = n-1.$

Worst-case time complexity (Mergesort)

- Basic operation: the comparison that takes place in *merge*.
- Input size: *n*, the number of items in array *S*.
- Time complexity:

  $W(n) = W(h) + W(m) + n \quad =>$

$$\begin{cases} W(n) = 2W(\frac{n}{2}) + n - 1 & for \quad n > 1, \quad n \quad a \quad power \quad of \quad 2 \\ W(1) = 0 \end{cases}$$

- Solution: $W(n) = n\lg n - (n - 1) \in \Theta(n\lg n)$

Memory complexity

- Memory complexity of Algorithm 2.2:

  $n(1 + 1/2 + 1/4 + \ldots) = 2n$
- To reduce the memory complexity to n:
- Algorithm 2.4: Mergesort 2
  - Problem: Sort n keys in nondecreasing sequence.
  - Inputs: positive integer n, array of keys S indexed from 1 to n.
  - Outputs: the array S containing the keys in nondecreasing order.

```
void mergesort2 (index low, index high)
{
    index mid;
    if (low < high) {
        mid = [(low + high)/2];
        mergesort2(low, mid);
        mergesort2(mid + 1, high);
        merge2(low, mid, high);
        }
}
```

Merge 2

- Algorithm 2.5: Merge2
  - Problem: Merge the two sorted subarrays of *S* created in Mergesort 2.
  - Inputs: indices *low, mid*, and *high*, and the subarray of *S* indexed from *low* to *high.* The keys in array slots from *low* to *mid* are already sorted in nondecreasing order, as are the keys in array slots from *mid* + 1 to *high.*
  - Outputs: the subarray of *S* indexed from *low* to *high* containing the keys in nondecreasing order.

```
void merge2 (index low, index mid, index high)
{
index i, j, k;
keytype U[low .. high]; // A local array needed for the merging
i = low; j = mid + 1; k = low;
while (i ≤ mid && j ≤ high){
        if (S[i]<S[j]){
                U[k] = S[i]; i++;}
        else {
                U[k] = S[j]; j++; }
        k++;
        }
if (i > mid)
        move S[j] through S[high] to U[k] through U[high];
else
        move S[i] through S[mid] to U[k] through U[high];
move U[low] through U[high] to S[low] through S[high];
}
```

Quicksort
جستجوی سریع

- در سال ۱۹۶۲توسط Hoare ابداع شد.
- مراحل:
  - یک عنصر به عنوان محور در نظر گرفته می‌شود (pivot). (معمولاً اولین عنصر)
  - آرایه با قرار دادن همه موارد کوچکتر از محور قبل از آن و همه موارد بزرگتر از محور بعد از آن به دو پارتیشن تقسیم می‌شود.
  - هر پارتیشن را به صورت بازگشتی مرتب می‌شود.

An example

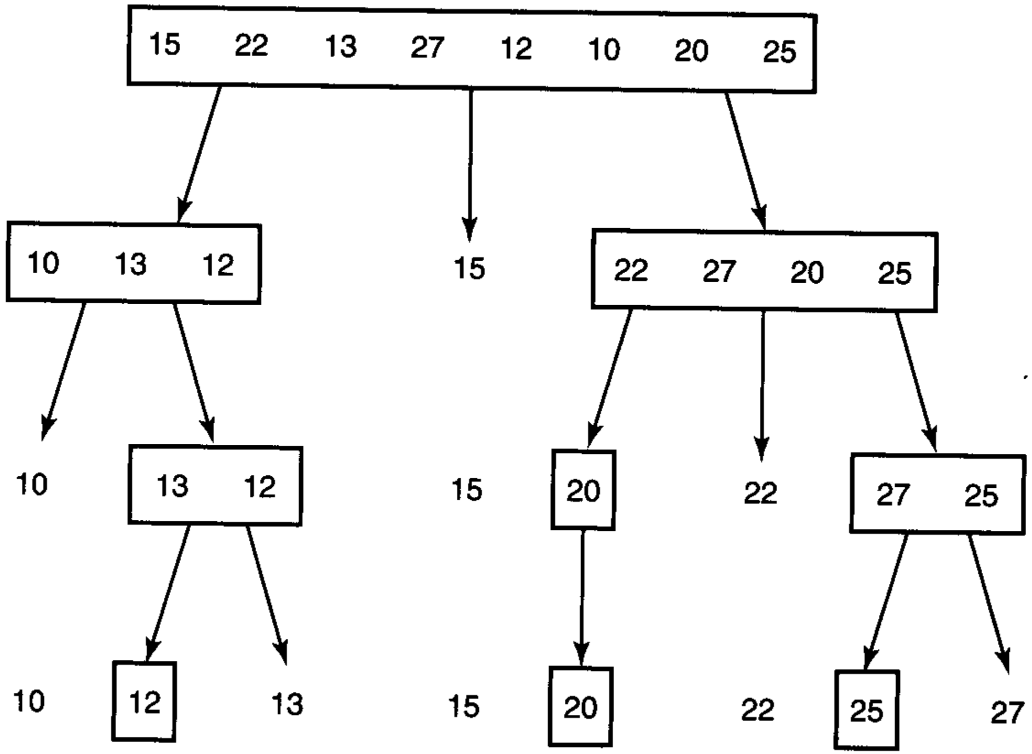


Figure 2.3 ● The steps done by a human when sorting with Quicksort. The subarrays are enclosed in rectangles whereas the pivot points are free.

The algorithm

- Algorithm 2.6: Quicksort
  - Problem: Sort *n* keys in nondecreasing order.
  - Inputs: positive integer *n*, array of keys *S* indexed from 1 to *n.*
  - Outputs: the array *S* containing the keys in nondecreasing order.

```
void quicksort (index low, index high){
    index pivotpoint;
    if (high > low){
            partition(low, high, pivotpoint);
            quicksort(low, pivotpoint - 1);
            quicksort(pivotpoint + 1, high);
    }
}
```

Partition

- Algorithm 2.7: Partition
  - Problem: Partition the array *S* for Quicksort.
  - Inputs: two indices, *low* and *high*, and the subarray of *S* indexed from *low* to *high.*
  - Outputs: *pivotpoint*, the pivot point for the subarray indexed from *low* to *high.*

```
void partition (index low, index high, index& pivotpoint) {
      index i, j;
      keytype pivotitem;
      pivotitem = S[low]; // Choose first item for pivotitem.
      j = low;
      for (i = low + 1; i <= high; i++)
            if (S[i] < pivotitem){
                  j++;
                  exchange S[i] and S[j];
            }
      pivotpoint = j;
      exchange S[low] and S[pivotpoint];
                  // Put pivotitem at pivotpoint.
      }
```

An example of procedure partition

Table 2.2: An example of procedure *partition*[*]

i	*j*	*S*[1]	*S*[2]	*S*[3]	*S*[4]	*S*[5]	*S*[6]	*S*[7]	*S*[8]	
—	—	15	22	13	27	12	10	20	25	← Initial values
2	1	**15**	**22**	13	27	12	10	20	25	
3	2	**15**	22	**13**	27	12	10	20	25	
4	2	**15**	13	22	**27**	12	10	20	25	
5	3	**15**	13	22	27	**12**	10	20	25	
6	4	**15**	13	12	27	22	**10**	20	25	
7	4	**15**	13	12	10	22	27	**20**	25	
8	4	**15**	13	12	10	22	27	20	**25**	
—	4	10	13	12	15	22	27	20	25	← Final values

[*]Items compared are in boldface. Items just exchanged appear in squares.

Every-case time complexity (Partition)

- Basic operation: the comparison of $S[i]$ with *pivotitem*.
- Input size: $n = high - low + 1$, the number of items in the subarray
- Time complexity: $T(n) = n - 1$

Worst-case time complexity (Quicksort)

- Basic operation: the comparison of *S*[*i*] with *pivotitem* in *partition*
- Input size: *n*, the number of items in the array *S*.
- Time complexity:

$$T(n) = \underbrace{T(0)}_{Time\ to\ sort\ left\ subarray} + \underbrace{T(n-1)}_{Time\ to\ sort\ right\ subarray} + \underbrace{n-1}_{Time\ to\ partition}$$

$$\begin{cases} T(n) = T(n-1) + n - 1 & for \quad n > 0 \\ T(0) = 0 \end{cases}$$

- Solution: $T(n) = n(n - 1)/2 \in \Theta(n^2)$

Average-case time complexity (Quicksort)

- Basic operation: the comparison of *S*[*i*] with *pivotitem* in *partition*
- Input size: *n*, the number of items in the array *S*.
- Time complexity:

$$A(n) = \sum_{p=1}^{n} \overbrace{\frac{1}{n}}^{\substack{\text{Probability} \\ \text{pivotpoint is } p}} \underbrace{[A(p-1) + A(n-p)]}_{\substack{\text{Average time to} \\ \text{sort subarrays when} \\ \text{pivotpoint is } p}} + \underbrace{n-1}_{\substack{\text{Time to} \\ \text{partition}}}$$

- Solution: $A(n) \approx 1.38(n + 1)\lg n \in \Theta(n\lg n)$

Arithmetic with large integers
حساب با اعداد بزرگ

- Representation of large integers: addition and other linear-time operations
  - use an array of integers
  - reserve the high-order array slot for the sign
  - linear-time algorithms
    - addition
    - subtraction
    - $u \times 10^m$
    - u divide 10^m
    - u rem10^m

Multiplication of large integers

- Split an n-digit integer into two integers of approximately $n/2$ digits, e.g.,
  - $567{,}832 = 567 \times 10^3 + 832$
  - $9{,}423{,}723 = 9423 \times 10^3 + 723$

- In general

$$\underbrace{u}_{n \text{ digits}} = \underbrace{x}_{\lceil n/2 \rceil \text{ digits}} \times 10^m + \underbrace{y}_{\lfloor n/2 \rfloor \text{ digits}}$$

where $m = \left\lfloor \frac{n}{2} \right\rfloor$

Multiplication

- To multiply two n-digit integers

  $u = x \times 10^m + y$

  $v = w \times 10^m + z$

- The product:

  $uv = xw \times 10^{2m} + (xz + wy) \times 10^m + yz$

- Example

  $567{,}832 \times 9{,}423{,}723 = (567 \times 10^3 + 832)(9423 \times 10^3 + 723) = \ldots ?$

The algorithm

- Algorithm 2.9: Large Integer Multiplication
  - Problem: Multiply two large integers, *u* and *v*.
  - Inputs: large integers *u* and *v*.
  - Outputs: *prod,* the product of *u* and *v*.

```
large_integer prod (large_integer u, large_integer v){
      large_integer x, y, w, z;
      int n, m; n = maximum (number of digits in u, number of digits in v)
      if (u = = 0 || v = = 0)
                return 0;
      else if (n <= threshold)
                return u × v obtained in the usual way;
      else {
                m = [n/2];
                x = u divide 10m;
                y = u rem 10m;
                w = v divide 10m;
                z = v rem 10m;
                return prod(x,w) x 10^2m + (prod(w,y)+prod(x,z)) x 10^m +
prod(y,z);
      }
}
```

Worst-case time complexity

- Basic operation: The manipulation of one decimal digit in a large integer when adding, subtracting, or doing *divide* 10^m, *rem* 10^m, or $\times 10^m$
- Input size: n, the number of digits in each of the two integers
- Time complexity

$$\begin{cases} W(n) = 4W(\frac{n}{2}) + cn & for \quad n > s, \quad n \quad a \quad power \quad of \quad 2 \\ W(s) = 0 \end{cases}$$

- Solution: $W(n) \in \Theta(n^{\lg 4}) = \Theta(n^2)$

Reduce the number of multiplications

- *prod* must determine: xw, $xz + yw$, and yz
- *prod* is called 4 times to calculate: xw, xz, yw, and yz
- Set $r = (x + y)(w + z) = xw + (xz + yw) + yz$

  then $xz + yw = r - xw - yz$

  We only need to compute:

  $r = (x + y)(w + z)$, xw, and yz

New algorithm

- Algorithm 2.10: Large Integer Multiplication 2
  - Problem: Multiply two large integers, *u* and *v*.
  - Inputs: large integers *u* and *v*.
  - Outputs: *prod2*, the product of *u* and *v*.

```
large_integer prod2 (large_integer u, large_integer v) {
        large_integer x, y, w, z, r, p, q;
        int n, m;
        n = maximum (number of digits in u, number of digits in v);
        if (u == 0 || v == 0)
                return 0;
        else if (n <= threshold)
                return u × v obtained in the usual way;
        else {
                m = [n/2];
                x = u divide 10^m; y = u rem 10^m;
                w = v divide 10^m; z = v rem 10^m;
                r = prod2(x + y, w + z);
                p = prod2(x, w);
                q = prod2(y, z);
                return p × 10^2m + (r-p-q) × 10^m+q;
        }
}
```

Worst-case time complexity

- Basic operation: The manipulation of one decimal digit in a large integer when adding, subtracting, or doing *divide* 10^m, *rem* 10^m, or $\times\ 10^m$
- Input size: *n*, the number of digits in each of the two integers
- Time complexity

$$\begin{cases} 3W(\frac{n}{2}) + cn \le W(n) \le 3W(\frac{n}{2}+1) + cn & \textit{for} \quad n > s, \quad n \quad a \quad power \quad of \quad 2 \\ W(s) = 0 \end{cases}$$

- Solution: $W(n) \in \Theta(n^{\lg 3}) = \Theta(n^{1.58})$

When not to use divide-and-conquer
در چه مسائلی روش تقسیم و حل مناسب نیست

If possible, we should avoid divide-and-conquer in the following two cases:

- An instance of size *n* is divided into two or more instances each almost of size *n*
  - Time complexity = exponential
  - e.g. Fibonacci sequence
- An instance of size *n* is divided into almost *n* instances of size *n*/*c*, where c is a constant
  - Time complexity = $n^{\Theta(\log n)}$

- Sometimes, a problem requires exponentiality
  - e.g. Consider the Towers of Hanoi problem

Exercises

- 2, 6
- 11
- 14, 15, (18)
- 19, 24
- 25
- 30, 32
- 35