Chapter 2:
Divide-and-Conquer

Contents

Control Abstraction

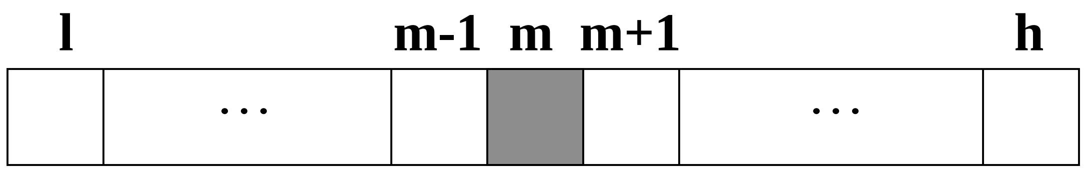
```
Type D&C (P){
   if small(P) return solution(P)
   else {
      divide P into smaller instances P_1, P_2, ... , P_k  k >= 1
      /* apply D&C to P_i
      return combine (D&C(P_1), ... , D&C(P_k));
   }
}
```

- **Let n_i be the size of P_i**

$$T(n) = \begin{cases} g(n) & \textbf{n small} \\ T(n_1) + \cdots + T(n_k) + \underbrace{f(n)}_{\text{divide \& combine time}} & \textbf{otherwise} \end{cases}$$

2.1 Binary Search

- **Informal description: given x**



- **See ex 2.1 and Fig. 2.1**
- **See Alg. 2.1: recursive binary search**

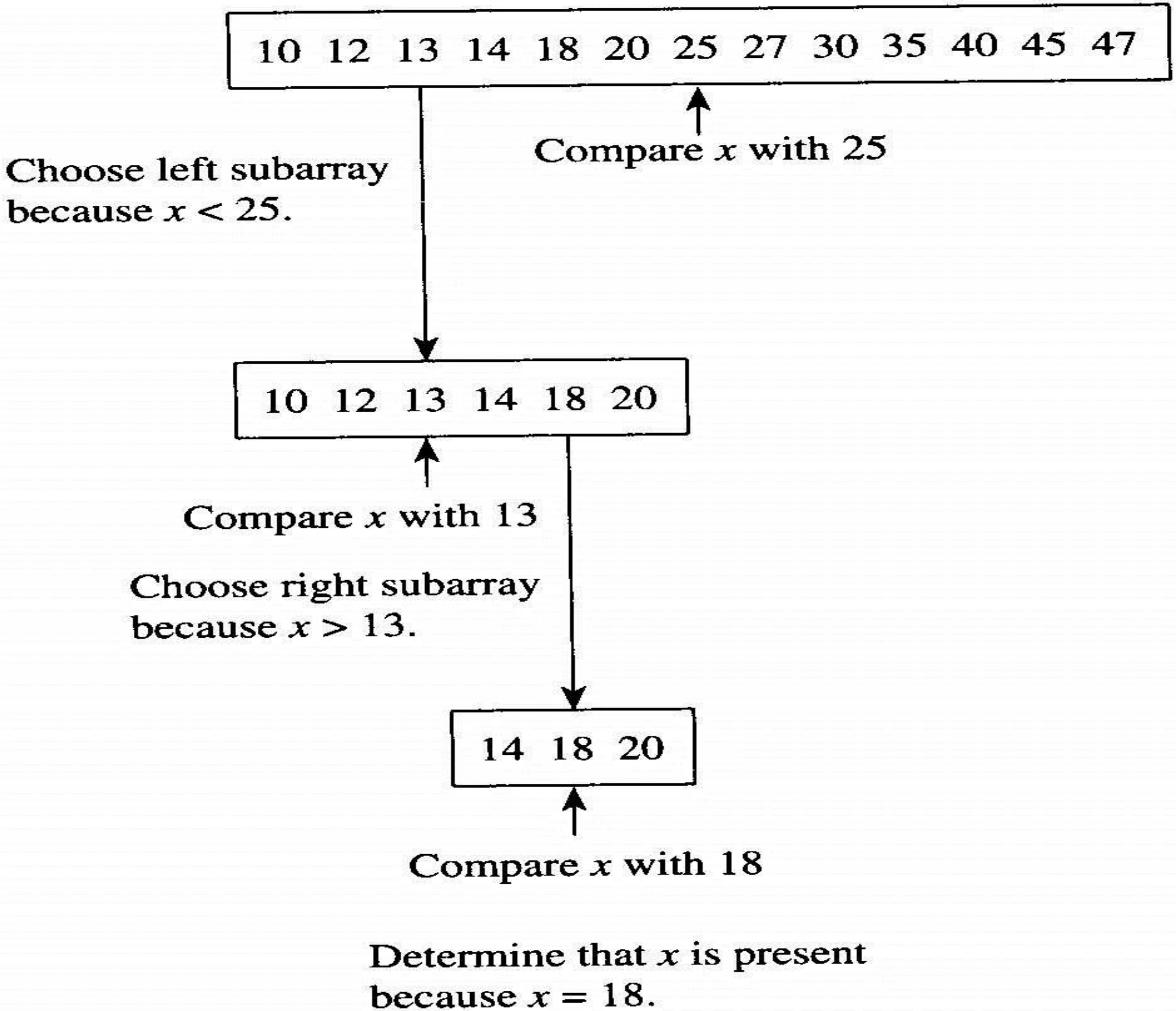


Figure 2.1 The steps done by a human when searching with Binary Search. (*Note:* $x = 18$.)

Algorithm 2.1 Binary Search

- **Problem: Determine whether *x* in the sorted array *S* of size *n*.**
- ***Inputs*: positive integer *n*, sorted (nondecreasing order) array of keys *S* indexed from 1 to *n*, and 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 analysis

- **Assume**

  1. $n = 2^k$

  2. One comparison (using efficient assembler)

$$W(1) = 1$$

$$
\begin{aligned}
W(n) &= W(n/2) + 1 \\
&= W(n/2^2) + 1 + 1 \\
&= W(n/2^3) + 1 + 1 + 1 \\
&\ \ \vdots \\
&= W(n/2^k) + \underbrace{1 + 1 + \cdots + 1}_{k} \\
&= 1 + \log n \in \Theta(\log n)
\end{aligned}
$$

(for proof by induction

→ See ex B.1)

2.2 Mergesort

1. **Divide the array into two subarrays**
2. **Conquer each subarray by sorting it (may use recursion)**
3. **Combine the two sorted subarrays by merging**

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

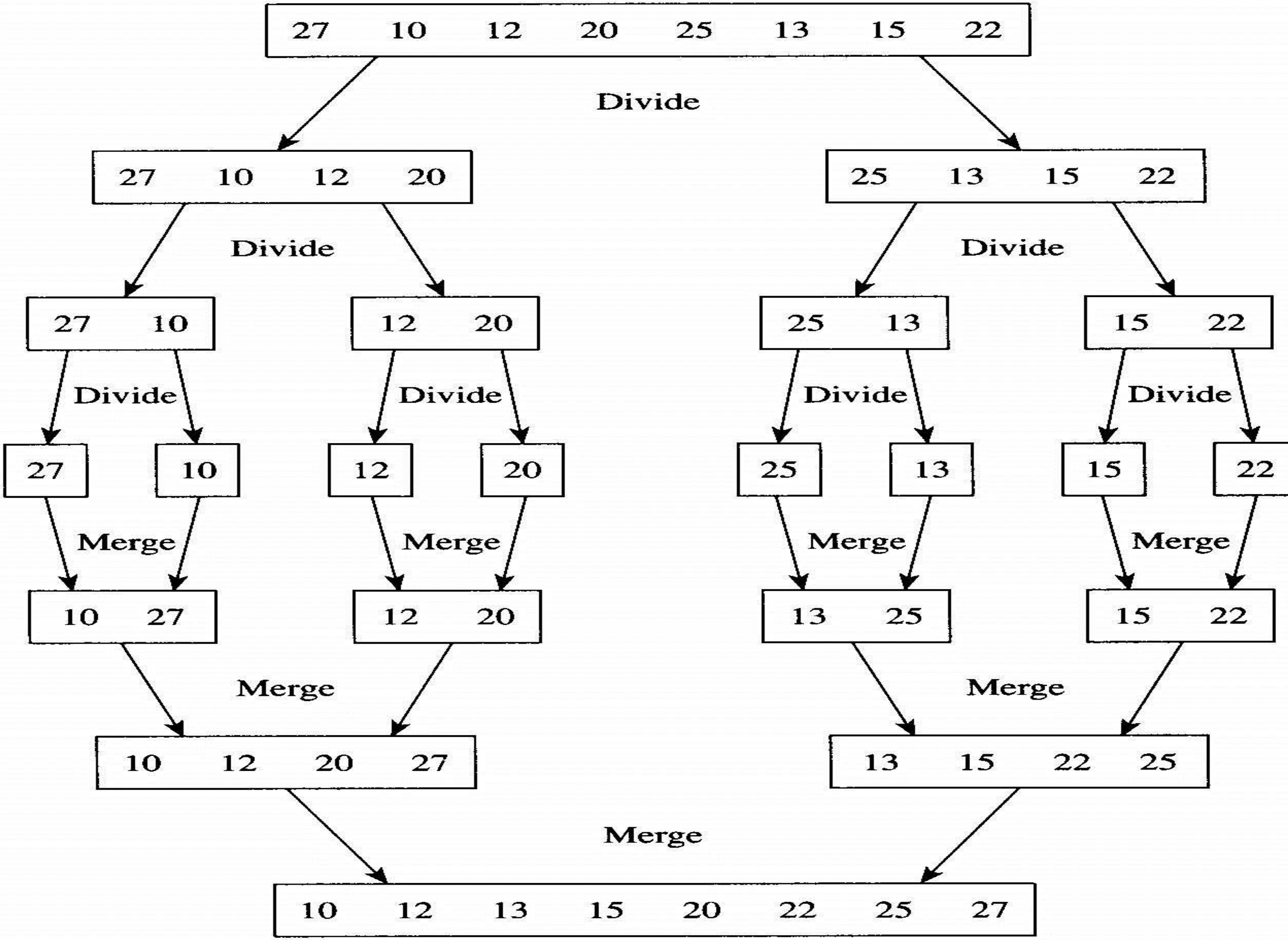


Table 2.1 An example of merging two arrays U and V into one array S*

k	U	V	S (Result)
1	**10** 12 20 27	**13** 15 22 25	10
2	10 **12** 20 27	**13** 15 22 25	10 12
3	10 12 **20** 27	**13** 15 22 25	10 12 13
4	10 12 **20** 27	13 **15** 22 25	10 12 13 15
5	10 12 **20** 27	13 15 **22** 25	10 12 13 15 20
6	10 12 20 **27**	13 15 **22** 25	10 12 13 15 20 22
7	10 12 20 **27**	13 15 22 **25**	10 12 13 15 20 22 25
—	10 12 20 27	13 15 22 25	10 12 13 15 20 22 25 27 ← Final values

*The items compared are in boldface.

Algorithm 2.4 Mergesort 2

- <u>Problem</u>: Sort *n* keys in nondecreasing sequence.
- <u>Inputs</u>: positive integer *n*, array of keys *S* indexed from 1 to *n*.
- <u>Outputs</u>: 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);
    }
}
```

Algorithm 2.5 Merge 2 (1/2)

- **<u>Problem</u>: Merge the two sorted subarrays *S* of created in Mergesort 2.**
- **<u>Inputs</u>: 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*.**
- **<u>Outputs</u>: the subarray of *S* indexed from *low* to *high* containing the keys in nondecreasing order.**

Algorithm 2.5 Merge 2 (2/2)

```
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];
}
```

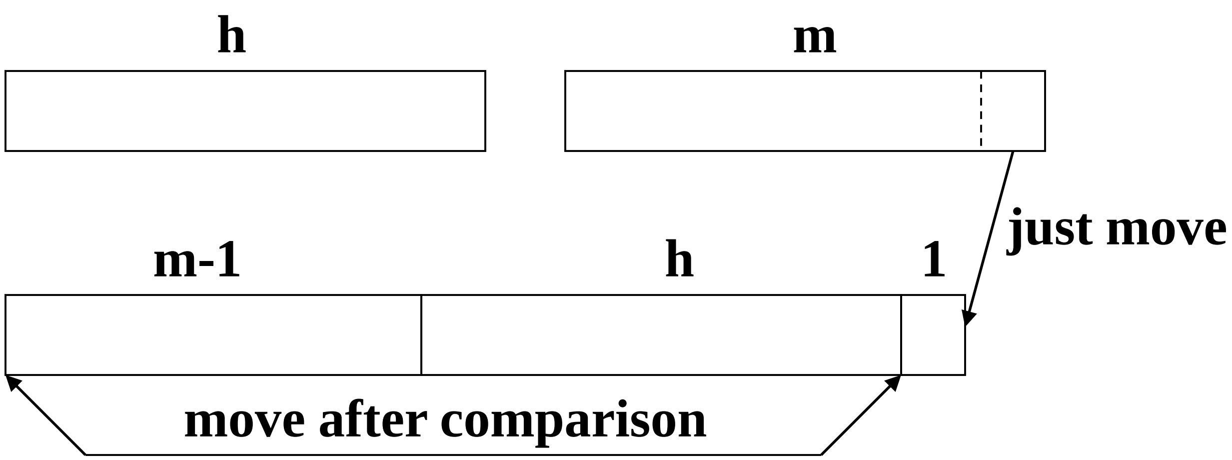
Merge 2(l, m, h)

```
      l              m   m+1             h
    [10   12   20   27 | 13   15   22   25]
      i →                 j →
    [10     {auxiliary global array}     ]
      k →
```

Time complexity analysis of merge2

- **Basic operation: comparison**
- **Input size: *h* and *m* (# of items in each array)**

$$W(h,m) = h + m - 1 \in \Theta(h+m)$$



- **See Alg. 2.4 mergesort 2**

Time complexity analysis of mergesort 2

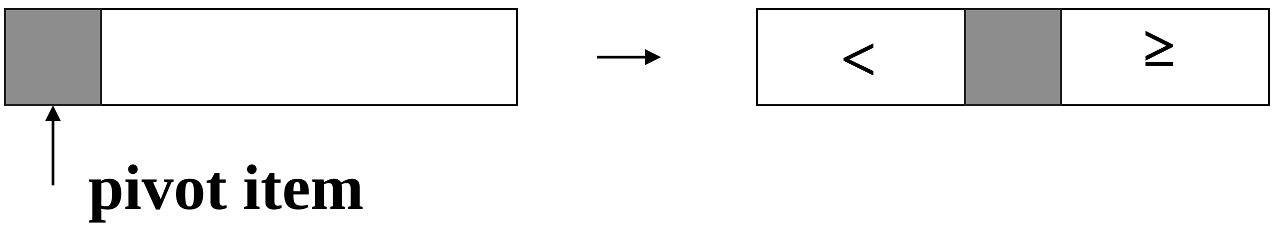
- **Basic operation: comparison**
- **Input size: n**
- **W(n) = W(h) +W(m) + (h + m – 1)**
- **Assume h = m = n/2,** $n = 2^k$

$$
\begin{aligned}
W(n) &= 2W(n/2) + n - 1 \\
&= 2[2W(n/2^2) + n/2 - 1] + n - 1 \\
&= 2^2 W(n/2^2) + 2n - (1+2) \\
&= 2^2[2W(n/2^3) + n/2^2 - 1] + 2n - (1+2) \\
&= 2^3 W(n/2^3) + 3n - (1+2+2^2) \\
&\vdots \\
&= 2^k W(n/2^k) + kn - (1+2+\cdots+2^{k-1}) \\
&= n\log n - (n-1) \in \Theta(n\log n)
\end{aligned}
$$

- **Space complexity**

  $S(n) \in \Theta(n)$

2.4 Quicksort (Hoare 1962)



- **See ex 2.3 and Fig. 2.3**
- **See Alg. 2.6 quicksort**
- **See Alg. 2.7 partition and Table 2.2**

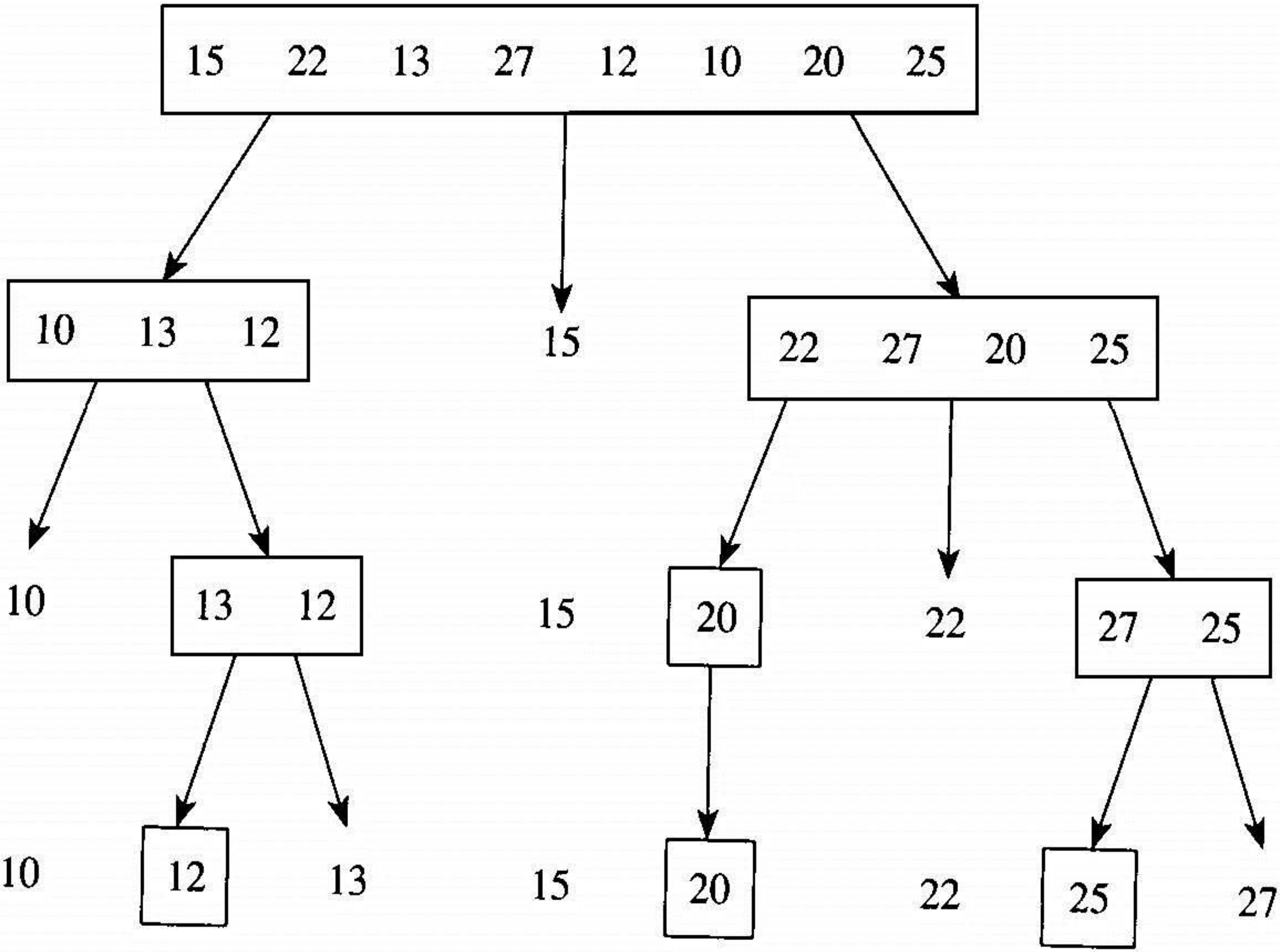


Figure 2.3 The steps done by a human when sorting with Quicksort. The subarrays are enclosed in rectangles whereas the pivot points are free.

Algorithm 2.6 Quicksort

- **<u>Problem</u>: Sort *n* keys in nondecreasing order.**
- **<u>Inputs</u>: positive integer *n*, array of keys *S* indexed from 1 to *n*.**
- **<u>Outputs</u>: 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);
   }
}
```

Algorithm 2.7 Partition (1/2)

- **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 of S indexed from *low* to *high*.**

Algorithm 2.7 Partition (2/2)

```
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++;  // Index for the smaller items than the pivot item.
            exchange S[i] and S[j];
        }
    pivotpoint = j;
    exchange S[low] and S[pivotpoint]; // Put pivotitem at pivotpoint.
}
```

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**	$\boxed{13}$	$\boxed{22}$	**27**	12	10	20	25	
5	3	**15**	13	22	27	**12**	10	20	25	
6	4	**15**	13	$\boxed{12}$	27	$\boxed{22}$	**10**	20	25	
7	4	**15**	13	12	$\boxed{10}$	22	$\boxed{27}$	**20**	25	
8	4	**15**	13	12	10	22	27	20	**25**	
—	4	$\boxed{10}$	13	12	$\boxed{15}$	22	27	20	25	←Final values

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

Time complexity analysis

- **T(n) of partition**
  - **Basic operation: comparison of S[i] with pivot item**
  - **Input size: n = h – l +1**
  - **T(n) = n – 1 →every item except the first is compared**
- **W(n) of quicksort**
  - **The worst case occurs if S is already sorted.**
  - **W(n) = W(0) + W(n-1) + n –1**

$$
\begin{aligned}
W(n) &= W(0) + W(n-1) + n - 1 \\
&= W(n-1) + (n-1) \\
&= W(n-2) + (n-2) + (n-1) \\
&\quad \ldots\ldots \\
&= W(0) + 0 + 1 + \ldots\ldots + (n-1) \\
&= n(n-1)/2 \in \Theta(n^2)
\end{aligned}
$$

A(n) of Quicksort (1/3)

- **Probability{pivotpoint is p} = 1/n**

$$A(n) = \sum_{p=1}^{n} \frac{1}{n}[A(p-1) + A(n-p)] + n - 1 \quad = \sum_{p=1}^{n} \frac{2}{n} A(p-1) + n - 1$$

- **multiplying by n** $\quad nA(n) = 2\sum_{p=1}^{n} A(p-1) + n(n-1) \cdots (1)$
- **substitute by n-1** $\quad (n-1)A(n-1) = 2\sum_{p=1}^{n-1} A(p-1) + (n-1)(n-2) \cdots (2)$
- **(1) – (2)** $\quad nA(n) - (n-1)A(n-1) = 2A(n-1) + 2(n-1)$

$$nA(n) - (n+1)A(n-1) = 2(n-1)$$

- **dividing by n(n+1)** $\quad \dfrac{A(n)}{n+1} = \dfrac{A(n-1)}{n} + \dfrac{2(n-1)}{n(n+1)} \qquad = a_n$

A(n) of Quicksort (2/3)

$$a_n = a_{n-1} + \frac{2(n-1)}{n(n+1)} \qquad\qquad a_n = \frac{A(n)}{n+1}$$

$$a_n = a_{n-1} + 2\left(\frac{2}{n+1} - \frac{1}{n}\right)$$

$$a_{n-1} = a_{n-2} + 2\left(\frac{2}{n} - \frac{1}{n-1}\right)$$

$$\vdots$$

$$a_2 = a_1 + 2\left(\frac{2}{3} - \frac{1}{2}\right)$$

$$+)\quad a_1 = a_0 + 2\left(\frac{2}{2} - \frac{1}{1}\right)$$

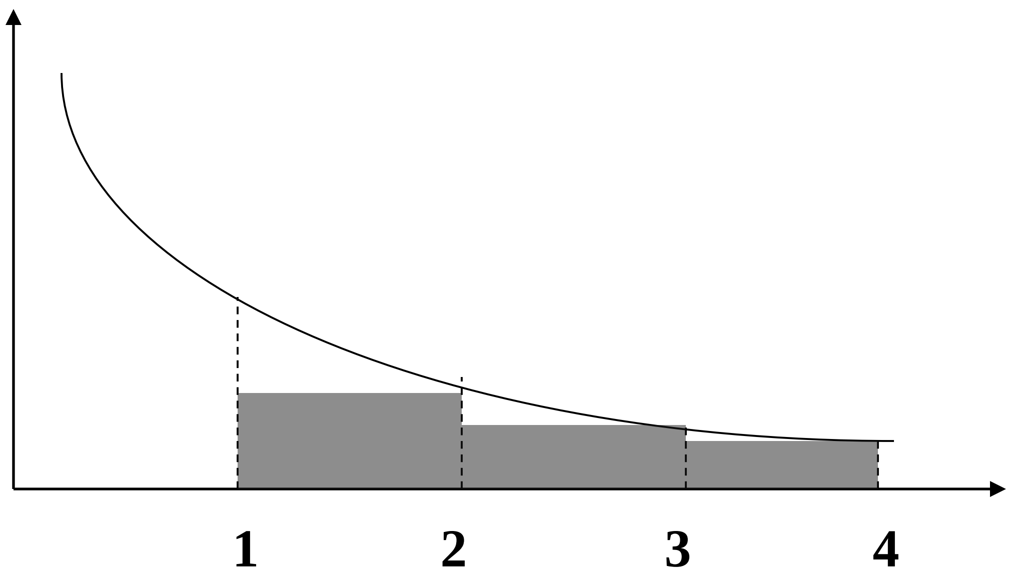
$$a_n = a_0 + \frac{4}{n+1} + 2\left(\frac{1}{n} + \frac{1}{n-1} + \cdots + \frac{1}{2}\right) - 2 \quad \approx 2\ln n$$

$$\therefore A(n) = (n+1)2\ln n$$

$$\in \Theta(n \log n)$$

A(n) of Quicksort (3/3)

$$S = \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$



$$S \le \int_1^n \frac{1}{x} dx = \ln n$$

2.5 Strassen's Matrix Multiplication Algorithm

```
for (i = 1; i <= n; i++)
   for (j = 1; j <= n; j++) {
      C[i][j] = 0;
      for (k = 1; k <= n; k++)
         C[i][j] = C[i][j] + A[i][k] * B[k][j];}
```

- **Alg. 1.4**
  - **# of multiplications** n^3
  - **# of addition** n^3 $(n^3 - n^2)$
- **Ex 2.4 2 x 2 case**

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

$$C = \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

$$\begin{aligned}
m_1 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\
m_2 &= (a_{21} + a_{22})b_{11} \\
m_3 &= a_{11}(b_{12} - b_{22}) \\
m_4 &= a_{22}(b_{21} - b_{11}) \\
m_5 &= (a_{11} + a_{12})b_{22} \\
m_6 &= (a_{21} - a_{11})(b_{11} + b_{12}) \\
m_7 &= (a_{12} - a_{22})(b_{21} + b_{22})
\end{aligned}$$

Strassen's Algorithm

- **Ex 2.4 2 x 2 case**
  - **General * 8 +/- 8**
  - **Strassen * 7 +/- 18**
  - **Assume** $n = 2^k$

- **See ex 2.5**
- **See Alg. 2.8**

$n \leq$ **threshold**	**standard**
O.W	**Strassen**

$$\begin{array}{c} \\ n/2 \end{array}\overset{\leftarrow n/2 \rightarrow}{\left[\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right]} = \left[\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right] \times \left[\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right]$$

Figure 2.4 The partitioning into submatrices in Strassen's Algorithm.

Strassen's Algorithm

- $\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{bmatrix}$ $\mathbf{B} = \begin{bmatrix} 8 & 9 & 1 & 2 \\ 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \end{bmatrix}$

$$\overbrace{\phantom{C_{11}}}^{2} \quad 2\left\{\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}\right. = \left[\begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ \hline 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{array}\right] \times \left[\begin{array}{cc|cc} 8 & 9 & 1 & 2 \\ 3 & 4 & 5 & 6 \\ \hline 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \end{array}\right]$$

Figure 2.5 • The partitioning in Strassen's algorithm with $n = 4$ and values

- **For example, C22 = M1 + M3 – M2 + M6**

Strassen's Algorithm

- **M1 = (A11+A22) x (B11+B22)**

$$= \left(\begin{bmatrix} 1 & 2 \\ 5 & 6 \end{bmatrix} + \begin{bmatrix} 2 & 3 \\ 6 & 7 \end{bmatrix}\right) x \left(\begin{bmatrix} 8 & 9 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 9 & 1 \\ 4 & 5 \end{bmatrix}\right)$$

$$= \begin{bmatrix} 3 & 5 \\ 11 & 13 \end{bmatrix} \mathbf{x} \begin{bmatrix} 17 & 10 \\ 7 & 9 \end{bmatrix} = \begin{bmatrix} 51+35 & 30+45 \\ 187+91 & 110+117 \end{bmatrix} = \begin{bmatrix} 86 & 75 \\ 278 & 227 \end{bmatrix}$$

- **Get M3, M2, M6 in the same way..**

C22 = M1 + M3 – M2 + M6

$$= \begin{bmatrix} 86 & 75 \\ 278 & 227 \end{bmatrix} + \begin{bmatrix} -6 & 3 \\ -34 & 11 \end{bmatrix} - \begin{bmatrix} 100 & 115 \\ 116 & 138 \end{bmatrix} + \begin{bmatrix} 64 & 78 \\ -17 & -21 \end{bmatrix}$$

$$= \begin{bmatrix} 44 & 41 \\ 111 & 79 \end{bmatrix}$$

Algorithm 2.8 Strassen

- <u>Problem</u>: Determine the product of two *n* x *n* matrices where *n* is a power of 2.
- <u>Inputs</u>: a positive integer *n* that is a power of 2, and two *n* x *n* matrices *A* and *B*.
- <u>Outputs</u>: the product *C* of *A* and *B*.

```
void strassen (int n, n x n_matrix A, n x n_matrix B, n x n_matrix& C) {
   if( n <= threshold )
      compute C = A x B using the standard algorithm;
   else {
      partition A into four submatrices A11, A12, A21, A22;
      partition B into four submatrices B11, B12, B21, B22;
      compute C = A x B using Strassen's Method;
         // example recursive call; strassen(n/2, A11+A22, B11+B22, M1)
   }
}
```

Time complexity analysis $n = 2^k$

- **T(n) of ***
  - **T(1) = 1**
  - **T(n) = 7T(n/2)**

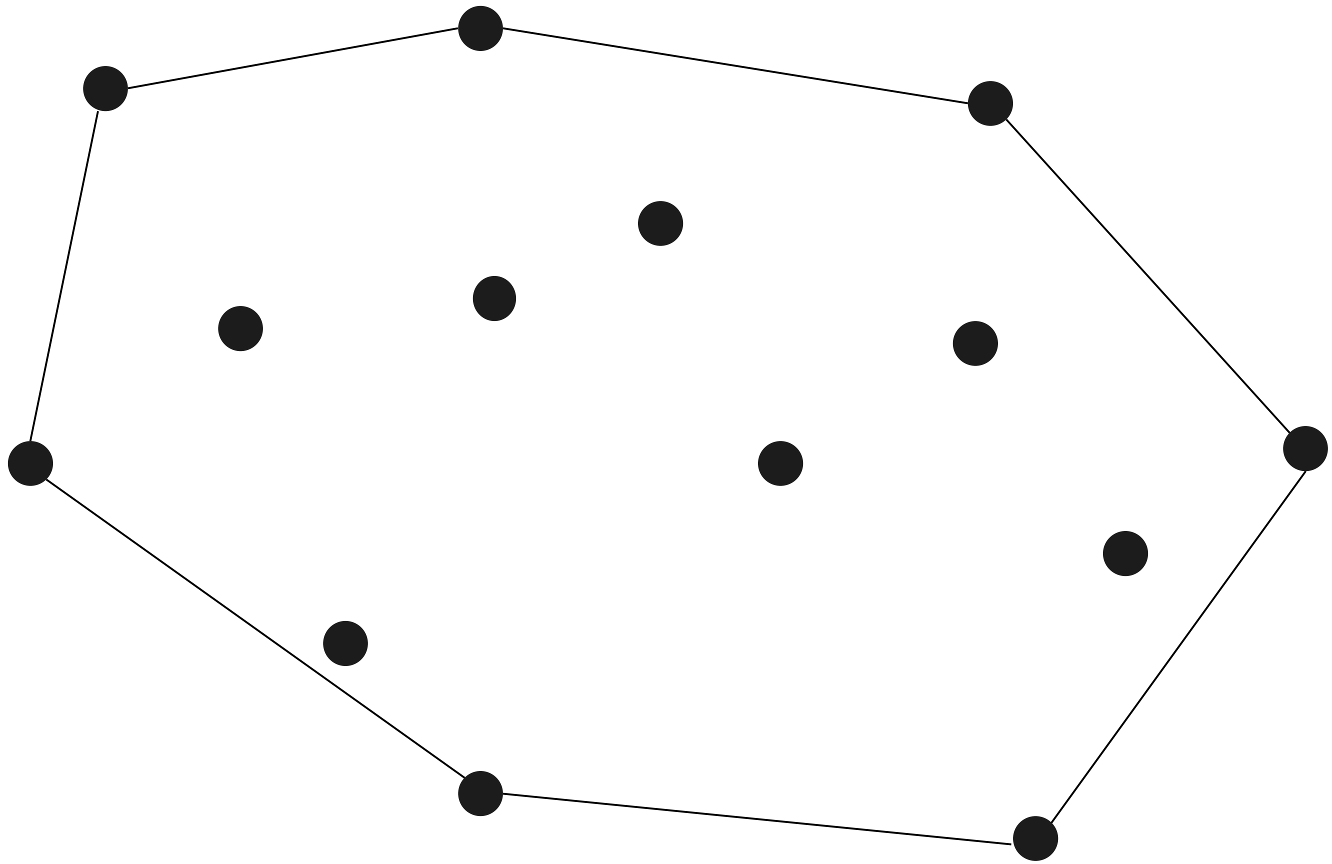
$$= 7 \cdot 7T(n/2^2)$$
$$\vdots$$
$$= 7^k T(n/2^k)$$
$$= 7^{\log n}$$
$$= n^{\log 7}$$
$$= n^{2.81}$$

- **T(n) of +**
  - **T(1) = 0**
  - **T(n) = 7T(n/2)** $+ 18(\frac{n}{2})^2$

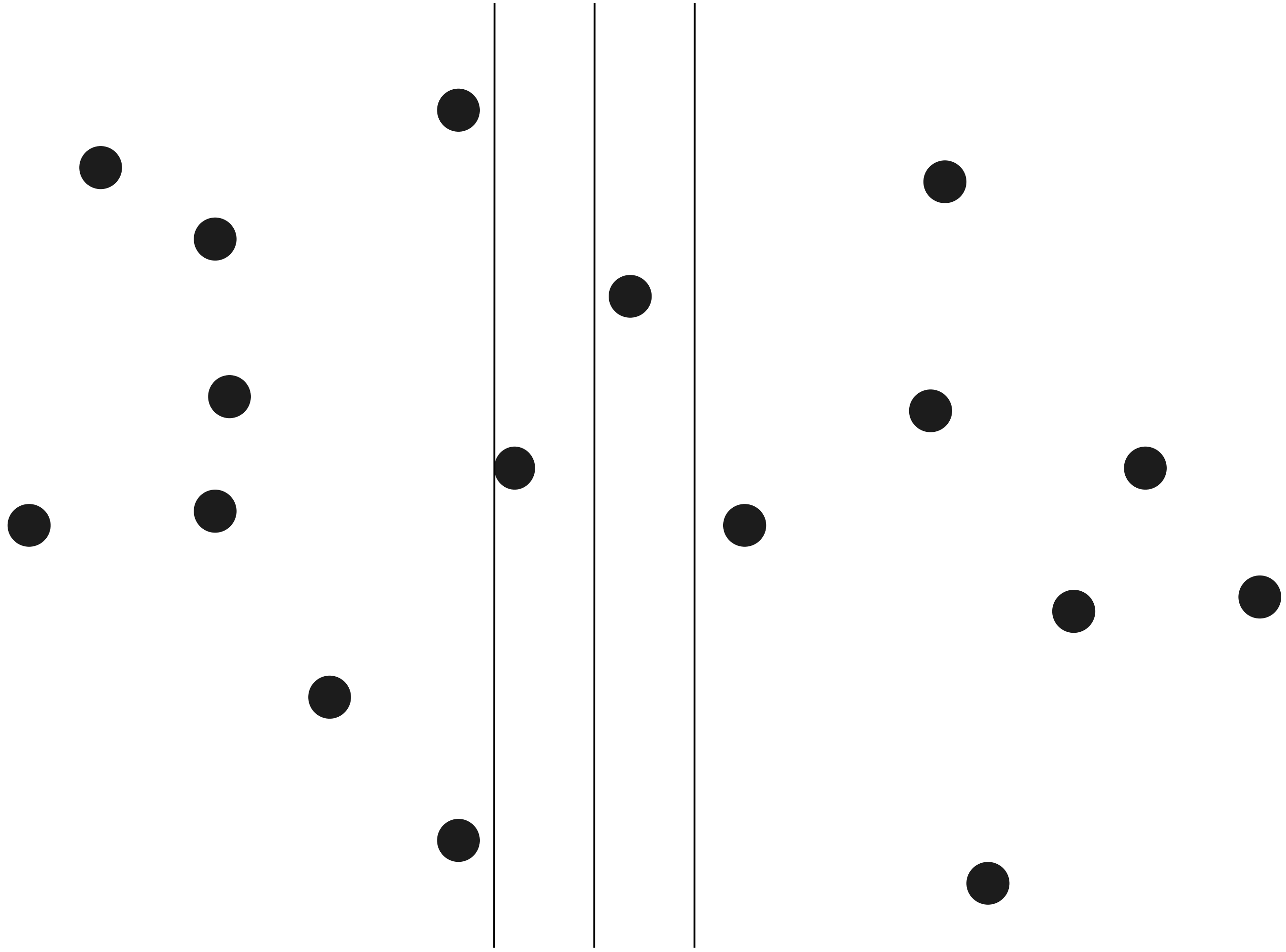
by ex B.20

$$T(n) = 6n^{\log 7} - 6n^2$$
$$\approx 6n^{2.81} - 6n^2$$
$$\in \Theta(n^{2.81})$$

Convex Hull



Closest Pair



Tree Drawing