[Chapter 2]

Divide-and-Conquer

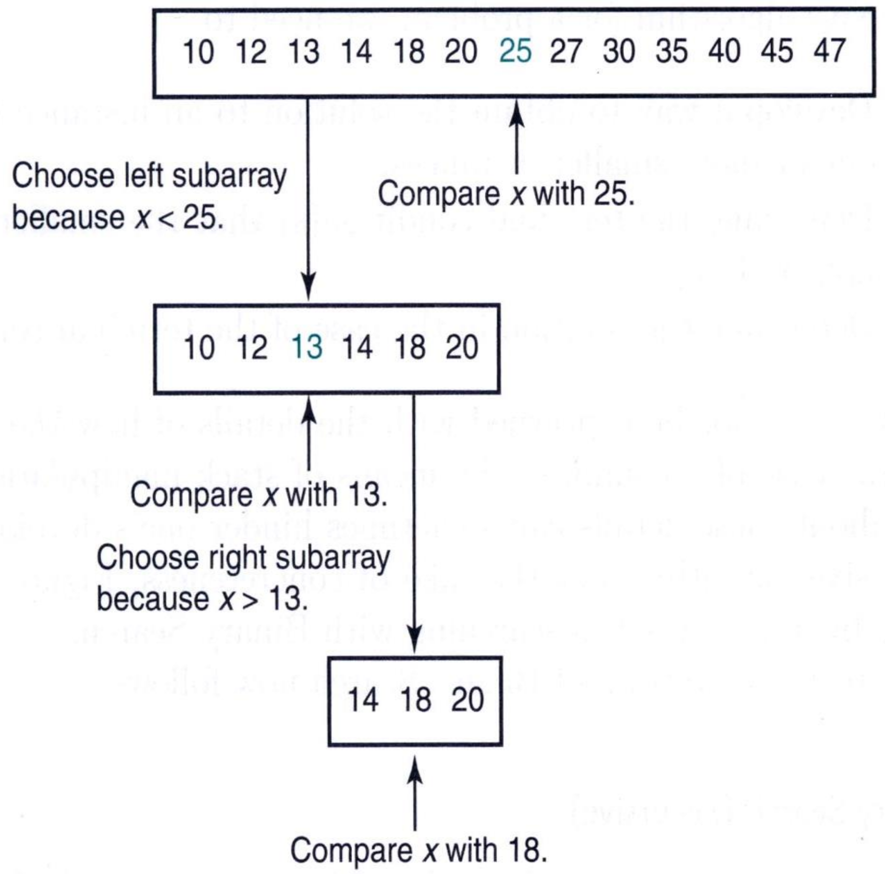
Divide-and-Conquer Approach

- Divide
  - It divides an instance of a problem into **two** or more smaller instances.
  - If the smaller instances are still too large to be solved readily, they can be divided into even smaller instances, until solutions are readily obtainable.
- Conquer
  - The smaller instances are usually instances of the original problem.
  - We may obtain solutions to the smaller instances readily.
- Combine
  - The process of dividing the instance can be obtained by combining these partial solutions.
- Top-down approach

Binary Search Algorithm Design

- Problem
  - Is the key x in the array S of n keys?
  - Determine whether x is in the sorted array S of n keys.
- Inputs (parameters)
  - Positive integer n, sorted (non-decreasing order) array of keys S indexed from 1 to n, a key x.
- Outputs
  - The location of x in S. (0 if x is not in S.)
- Design Strategy
  - *Divide* the array into two subarrays about half as large. If x is smaller than the middle item, choose the left subarray. If x is larger, choose the right one.
  - *Conquer* (solve) the subarray by determining whether x is in that subarray. Unless the subarray is sufficiently small, use recursion to do this.
  - *Obtain* the solution to the array from the solution to the subarray.

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



Binary Search (Recursive algorithm)

```
index location (index low, index high) {
      index mid;

      if (low > high)
          return 0;                              // Not found.
      else {
          mid = (low + high) / 2                 // Integer division.
          if (x == S[mid])
                return mid;                      //  Found.
          else if (x < S[mid])
                return location(low, mid-1);    // Choose the left sub-array.
          else
                return location(mid+1, high);   // Choose the right sub-array.
      }
}
...
locationout = location(1, n);
...
```

Points of Observation

- Reason for using a local variable *locationout*
  - Input parameters, n, S, x, will not be changed during running the algorithm.
  - Dragging those unchanging variables for every recursive call would incur a source of unnecessary inefficiency.
- Tail-recursion removal
  - No operations are done after the recursive call.
  - It is straightforward to produce an iterative version.
  - Recursion clearly illustrates the divide-and-conquer process.
  - However, running recursions is over-burdensome due to excessive uses of activation records.
  - A substantial amount of memory can be saved by eliminating the stack for activation records. (reason for preferring to iteration)
  - Iterative version is better only as constant factor. Order is same.

Worst-Case Time Complexity

- Basic operation: the comparison of *x* with *S[mid]*
- Input size: *n*, the number of items in the array.
- Case 1: When n is a power of 2.

$$W(n) = W(\tfrac{n}{2}) + 1$$

$$W(1) = 1$$

$$W(1) = 1$$

$$W(2) = W(1) + 1 = 2$$

$$W(4) = W(2) + 1 = 3$$

$$W(8) = W(4) + 1 = 4$$

$$W(16) = W(8) + 1 = 5$$

$$W(2^k) = k + 1$$

$$W(n) = \lg n + 1$$

WCTC: Proof for Case 1.

- Induction Base
  - When $n = 1$, $W(1) = 1 = \lg 1 + 1$.
- Induction Hypothesis
  - For some $n = 2^k (k \geq 1)$, suppose it holds that $W(n) = \lg n + 1$.
- Induction Step

$$\begin{aligned} W(2n) &= W(n) + 1 \\ &= \lg n + 1 + 1 \\ &= \lg n + \lg 2 + 1 \\ &= \lg(2n) + 1 \end{aligned}$$

WCTC: Case 2. When n is not a power of 2.

- $n = \lfloor \frac{n}{2} \rfloor$
  - The largest integer that is not greater than *n/2*.
  - This is the size of the half of the array.
  - $mid = \left\lfloor \frac{1+n}{2} \right\rfloor$

n	Size of the left sub-array	mid	Size of the right sub-array
Even	n/2 - 1	1	n/2
Odd	(n-1)/2	1	(n-1)/2

- Then, the WCTC can be represented by
  - $W(n) = 1 + W(\lfloor \frac{n}{2} \rfloor) \quad n > 1$

    $W(1) = 1$

WCTC: Case 2. Proof that $W(n)=\lfloor \lg n \rfloor+1$ (1/2)

- Induction Base
  - When *n* = 1, $\lfloor \lg n \rfloor+1=\lfloor \lg 1 \rfloor+1=0+1=1=W(1)$
- Induction Hypothesis
  - For some $n \geq 1$ and $1 < k < n$, suppose $W(k)=\lfloor \lg k \rfloor+1$
- Induction Step
  - if $\left\lfloor \frac{n}{2} \right\rfloor = \frac{n}{2}$

$$\begin{aligned}
W(n) &= 1+W(\lfloor \tfrac{n}{2} \rfloor) \\
&= 1+\lfloor \lg\lfloor \tfrac{n}{2} \rfloor \rfloor+1 \\
&= 2+\lfloor \lg\lfloor \tfrac{n}{2} \rfloor \rfloor \\
&= 2+\lfloor \lg \tfrac{n}{2} \rfloor \\
&= 2+\lfloor \lg n-1 \rfloor \\
&= 2+\lfloor \lg n \rfloor-1 \\
&= 1+\lfloor \lg n \rfloor
\end{aligned}$$

WCTC: Case 2. Proof that $W(n) = \lfloor \lg n \rfloor + 1$ (2/2)

- Induction Step (Cont'd)
  - if $\lfloor \frac{n}{2} \rfloor = \frac{n-1}{2}$

$$\begin{aligned}
W(n) &= 1 + W(\lfloor \tfrac{n}{2} \rfloor) \\
&= 1 + \lfloor \lg \lfloor \tfrac{n}{2} \rfloor \rfloor + 1 \\
&= 2 + \lfloor \lg \lfloor \tfrac{n}{2} \rfloor \rfloor \\
&= 2 + \lfloor \lg \tfrac{n-1}{2} \rfloor \\
&= 2 + \lfloor \lg(n-1) - 1 \rfloor \\
&= 2 + \lfloor \lg(n-1) \rfloor - 1 \\
&= 1 + \lfloor \lg(n-1) \rfloor \\
&= 1 + \lfloor \lg n \rfloor
\end{aligned}$$

Q.E.D.

Mergesort

- Problem
  - Sort n keys in nondecreasing order.
- Inputs (parameters)
  - Positive integer n, array of keys S indexed from 1 to n.
- Outputs
  - The array S containing the keys in nondecreasing order.
- Design Strategy
  - *Divide* the array into two subarrays each with $n/2$ items.
  - *Conquer* (solve) each subarray by sorting it. Unless the array is sufficiently small, use recursion to do this.
  - *Combine* the solutions to the subarrays by merging them into a single sorted array.

Mergesort Algorithm in Pseudo-code

```
void merge(int h, int m, const keytype U[], const keytype V[],
                keytype S[])
{
            index 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];
}
```

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

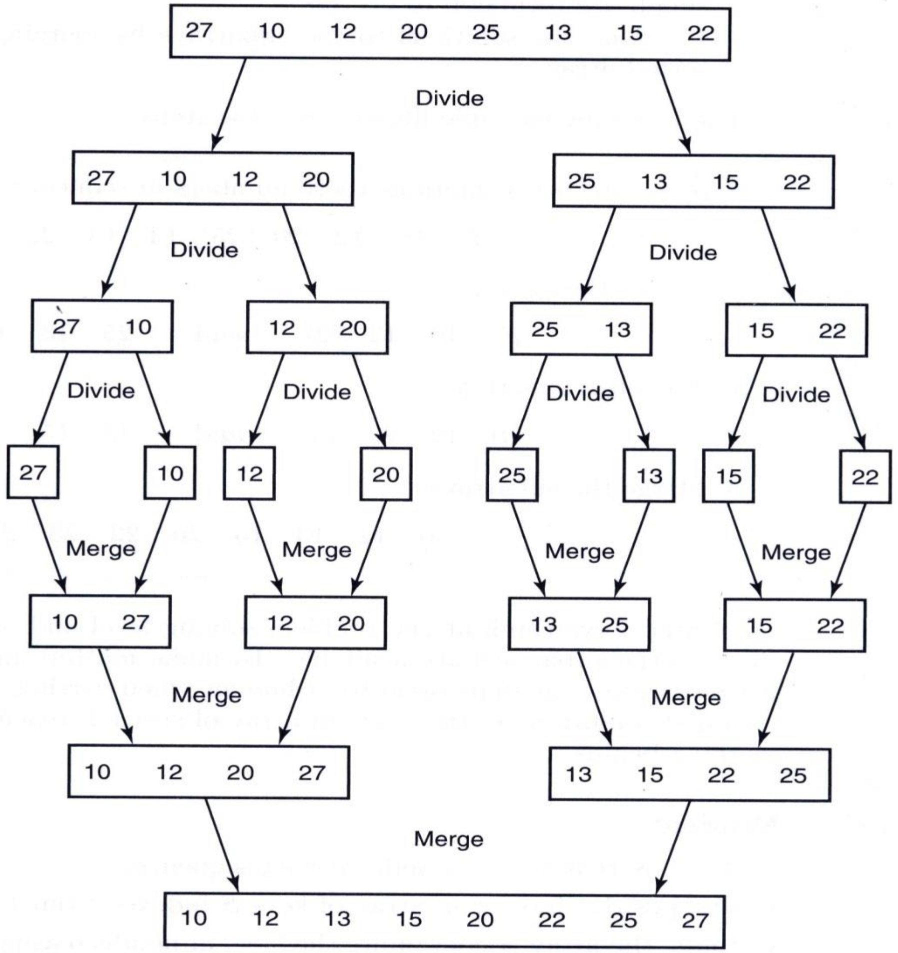
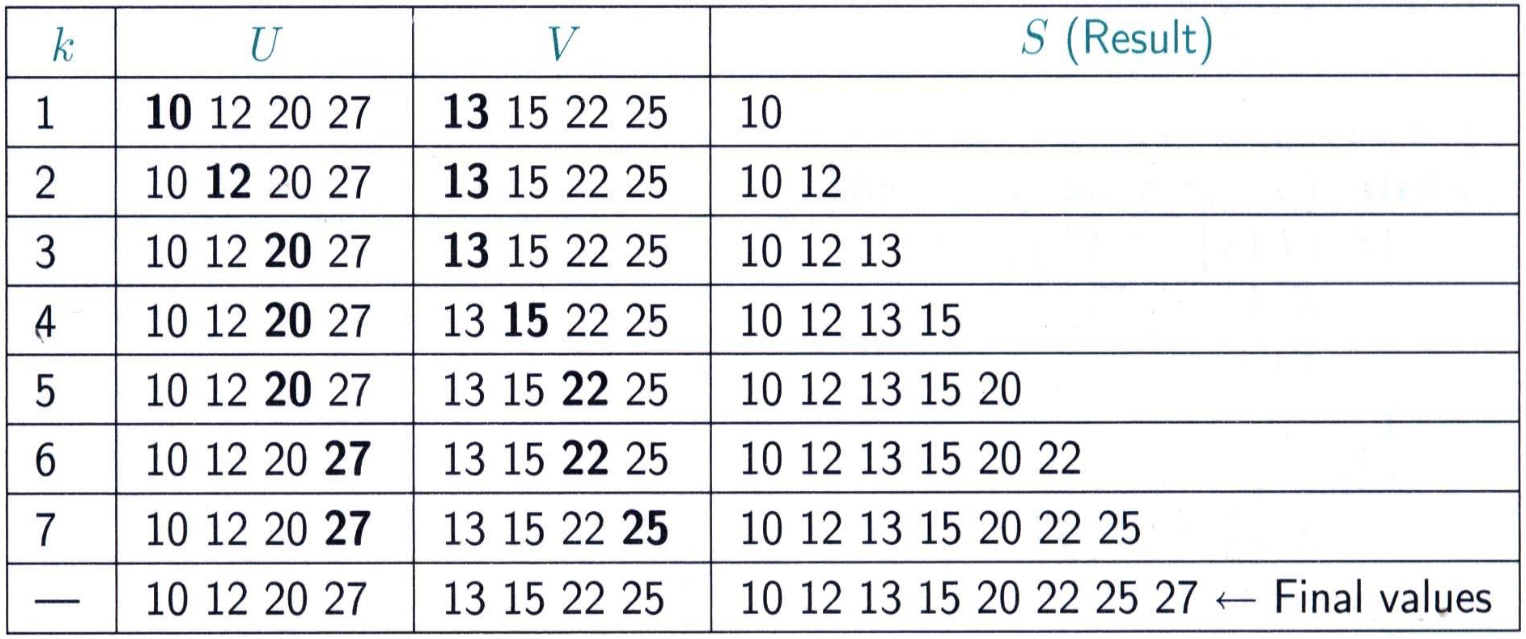


Table 2.1 An example of merging two arrays *U* and *V* into one array *S**



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

Worst-Case Time Complexity: Merge

- Basic operation: the comparison of ***U[i]*** and ***V[j]***.
- Input size: h, and m, the number of items in each of the two input arrays.
- Analysis:
  - Worst case: when $i = h$, and $j = m - 1$.
  - $W(h, m) = h + m - 1$.

Worst-Case Time Complexity: Mergesort

- Basic operation: the comparison that takes place in *merge*.
- Input size: *n*, the number of items in the array *S*.
- Analysis:
  - $W(h, m) = W(h) + W(m) + h + m - 1$.
    - $W(h)$ is the time to sort U.
    - $W(m)$ is the time to sort V/
    - $h + m - 1$ is the time to merge.
  - $W(n) = 2W(\frac{n}{2}) + n - 1 \qquad n > 1,\ n = 2^k\ (k \geq 1)$

    $W(1) = 0$
  - $W(n) = \Theta(n \lg n)$

Space Complexity Analysis

- In-place sort
  - A sorting algorithm that does not use any extra space beyond that needed to store the input.
  - Mergesort() is not an in-place sorting algorithm.
  - New arrays U and V will be created when *mergesort* is called.
  - The total number of extra array items created is $n+\frac{n}{2}+\frac{n}{4}+\cdots=2n$
  - In other words, the space complexity is

    $2n\in\Theta(n)$
  - We may reduce the extra space to n.
  - But it is not possible to make mergesort algorithm to be an in-place sort.

Mergesort2

```
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);
      }
}
…
mergesort2(1, n);
...
```

Merge2

```
void merge2(index low, index mid, index high) {
        index i=low, j=mid+1, k=low;   keytype U[low..high
        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)
           copy S[j] through S[high] to U[k] through U[high];
        else
           copy S[i] through S[mid] to U[k] through U[high];
        copy U[low] through U[high] to S[low] through S[high];
}
```

The Master Theorem

- Suppose a recurrence relation has a form of
  - $T(n) = a \times T\left(\frac{n}{b}\right) + f(n)$
  - *where* $a >= 1$ & $b > 1$ and n is a non-negative integer.
- Then, $T(n)$ has the following asymptotic bounds as follows.
  - For some constant $\varepsilon > 0$, if $f(n) = O\left(n^{\log_b a - \varepsilon}\right)$, $T(n) = \Theta\left(n^{\log_b a}\right)$
  - If $f(n) = \Theta\left(n^{\log_b a}\right)$, $T(n) = \Theta\left(n^{\log_b a} \lg n\right)$
  - For some constant $\varepsilon > 0$, if $f(n) = \Omega\left(n^{\log_b a + \varepsilon}\right)$, and if $a \times f\left(\frac{n}{b}\right) \leq c \times f(n)$ where there exist a positive constant $c < 1$ and sufficiently large n
    - $T(n) = \Theta(f(n))$

Examples: Master Theorem (1/3)

- $T(n) = 9T\left(\frac{n}{3}\right) + n$
  - $a = 9, b = 3, f(n) = n,\ n^{\log_b a} = n^{\log_3 9} = \Theta\left(n^2\right)$
  - So, $f(n) = O(n^{\log_3 9 - \varepsilon})$, where $\varepsilon = 1$.
  - Therefore, we can apply the Master Theorem #1
  - $T(n) = \Theta(n^{\log_3 9}) = \Theta(n^2)$
- $T(n) = T\left(\frac{2n}{3}\right) + 1$
  - $a = 1, b = 3/2,\ f(n) = 1$, and $n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = n^0 = \Theta(1)$
  - So, $f(n) = \Theta(1)$
  - Therefore, we can apply the Master Theorem #2
  - $T(n) = \Theta(1 \lg n) = \Theta(\lg n)$

Examples: Master Theorem (2/3)

- $T(n) = 3T\left(\frac{n}{4}\right) + n\lg n$
  - $a = 3$, $b = 4$, $f(n) = n \lg n$, and $n^{\log_b a} = n^{\log_4 3} = O\left(n^{0.793}\right)$
  - So, $f(n) = \Omega(n^{\log_4 3+\varepsilon})$, where $\varepsilon > 0$.
  - Let's see if we can apply the Master Theorem #3
  - For sufficiently large n, check if there exists c < 1.
  - See, if $c = \frac{3}{4}$, then $3\frac{n}{4}\lg(\frac{n}{4}) \le \frac{3}{4} n \lg n$ holds for sufficiently large n.
  - Therefore, $T(n) = \Theta(n \lg n)$

Examples: Master Theorem (3/3)

- $T(n) = 2T\left(\frac{n}{2}\right) + n \lg n$
  - $a = 2$, $b = 2$, $f(n) = n \lg n$, and $n^{\log_b a} = n^{\log_2 2} = \Theta(n)$
  - So, $f(n) = \Omega(n^{\log_2 2+\varepsilon})$, where $\varepsilon > 0$.
  - Let's see if we can apply the Master Theorem #3
  - For sufficiently large n, check if there exists c < 1, $2f\left(\frac{n}{2}\right) \leq c \times f(n)$
  - But, there exists no such c for sufficiently large n.
  - Because, $\frac{\lg n - 1}{\lg n} \leq c$, no matter how large c that is close to 1, we can always make the LHS less than c.
  - Therefore, we can't apply the Master Theorem #3.

Auxiliary Master Theorem

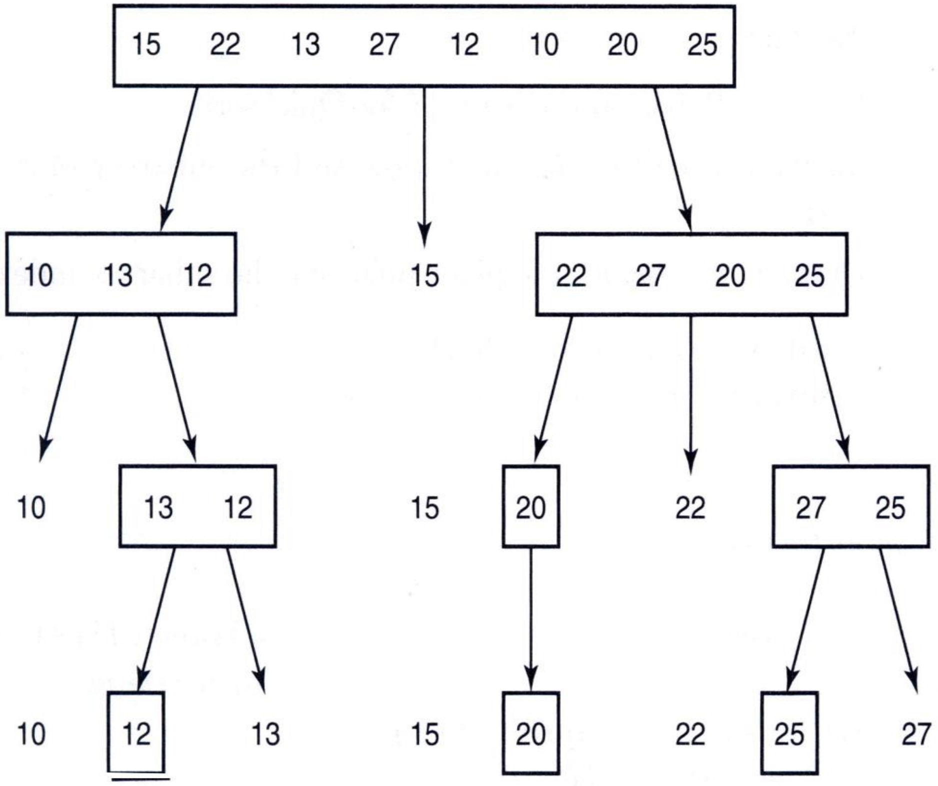
- $T(n) = a \times T\left(\frac{n}{b}\right) + f(n)$
  - For some $k \geq 0$, if $f(n) = \Theta(n^{\log_b a} \lg^k n)$
  - Then, $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
- Example: $T(n) = 2T\left(\frac{n}{2}\right) + n \lg n$
  - $f(n) = \Theta(n \lg n)$ where some $k=1$, $a=b=2$.
  - Therefore, $T(n) = \Theta(n \lg^2 n)$

Quicksort

- C. A. R. Hoare (1962)
- Partition Exchange Sort

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

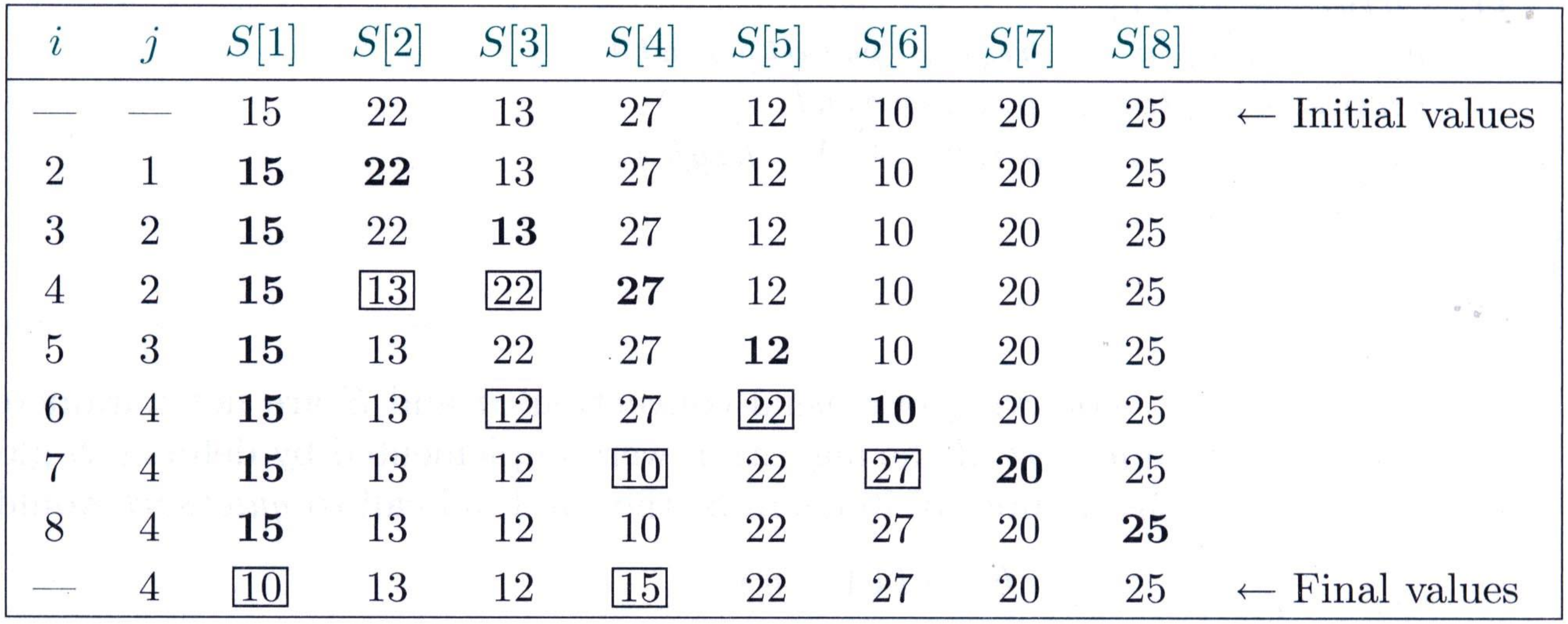
Figure 2.3 The steps done by a human when sorting with Quicksort. The subarrays are enclosed in rectangles whereas the pivot points are free.



Partitioning Algorithm

```
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
}
```

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

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 array *S*.
  - Because every item except the first is compared.
  - $T(n) = n - 1$.

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*.
  - The worst case occurs if the array is already sorted in nondecreasing order.
  - No items are less than the first item, if it is sorted.
  - Therefore, the array is repeatedly partitioned into an empty subarray on the left and a subarray with one less item on the right.
  - $T(n) = T(0) + T(n - 1) + n - 1.$
  - Since $T(0) = 0$, $T(n) = T(n - 1) + n - 1.$

Worst-Case Time Complexity (Quicksort) (Cont'd)

- Solving the recurrence relation,

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

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

$T(n - 2) = T(n - 3) + n - 3$

…

$T(2) = T(1) + 1$

$T(1) = T(0) + 0$

$T(0) = 0$

$$T(n) = 1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2}$$

Proof of $W(n) \le \frac{n(n-1)}{2}$

- Induction Base
  - If $n = 0$, $W(0) \le \frac{0(0-1)}{2}$.
- Induction Hypothesis
  - For all $0 \le k < n$, suppose $W(k) \le \frac{k(k-1)}{2}$
- Induction Step $W(n) \le \frac{n(n-1)}{2}$
  - $W(n) \le W(p-1) + W(n-p) + n - 1$

$$\le \frac{(p-1)(p-2)}{2} + \frac{(n-p)(n-p-1)}{2} + n - 1$$

$$= \frac{p^2 - 3p + 2 + (n-p)^2 - n + p + 2n - 2}{2}$$

$$= \frac{p^2 + (n-p)^2 + n - 2p}{2}$$

  - When $p = 1$, or $p = n$, the numerator will have the maximum value.

Proof of $W(n) \le \frac{n(n-1)}{2}$ (Cont'd)

- Therefore,
  - $max_{p=1}(p^2 + (n-p)^2 + n - 2p) = 1^2 + (n-1)^2 + n - 2 = n^2 - n$
  - $max_{p=n}(p^2 + (n-p)^2 + n - 2p) = n^2 + 0^2 + n - 2n = n^2 - n$
  - Consequently, the worst-case time complexity is

$$W(n) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Average-Case Time Complexity (Quicksort)

- Analysis
  - The value of pivotitem returned by partition is equally likely to be any of the numbers from 1 through n.
  - The probability for the pivot position to be the p-th is $\frac{1}{n}$
  - The average time to sort if the pivot position is the p-th is $[A(p - 1) + A(n - p)]$ and the time to partition is $n - 1$.
  - Therefore, the average time complexity is ...

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

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

Average-Case Time Complexity (Quicksort) (Cont'd)

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

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

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

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

$$a_n = \frac{A(n)}{n+1} \qquad a_n = a_{n-1} + \frac{2(n-1)}{n(n+1)} \quad n > 0$$

$$a_n = a_{n-1} + \frac{2(n-1)}{n(n+1)} \quad a_{n-1} = a_{n-2} + \frac{2(n-2)}{(n-1)n} \quad a_2 = a_1 + \tfrac{1}{3} \quad a_1 = a_0 + 0$$

Average-Case Time Complexity (Quicksort) (Cont'd)

$$
\begin{aligned}
a_n &= \sum_{i=1}^{n} \tfrac{2(i-1)}{i(i+1)} \\
&= 2\left(\sum_{i=1}^{n} \tfrac{1}{i+1} - \sum_{i=1}^{n} \tfrac{1}{i(i+1)}\right) \approx 2 \ln n
\end{aligned}
$$

$$\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \cdots + \frac{1}{n} = \ln n$$

$$
\begin{aligned}
A(n) &\approx (n+1) 2 \ln n \\
&= (n+1) 2 (\lg n) / (\lg e) \\
&\approx 1.38 (n+1) \lg n \\
&\in \Theta(n \lg n)
\end{aligned}
$$

Matrix Multiplication

```
void matrixmult (int n, const number A[][], const number B[][],
                         number C[][]) {
    index i, j, k;
    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];
        }
}
```

$$T(n) = n \times n \times n = n^3 \in \Theta(n^3)$$

2x2 Matrix Multiplication: Strassen's Method

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

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

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

*n*x*n* Matrix Multiplication: Strassen's Method

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

$$C = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 + M_3 - M_2 + M_6 \end{bmatrix}$$

$$M_1 = (A_{11} + A_{22}) \times (B_{11} + B_{22})$$
$$M_2 = (A_{21} + A_{22}) \times B_{11}$$
$$M_3 = A_{11} \times (B_{12} - B_{22})$$
$$M_4 = A_{22} \times (B_{21} - B_{11})$$
$$M_5 = (A_{11} + A_{12}) \times B_{22}$$
$$M_6 = (A_{21} - A_{11}) \times (B_{11} + B_{12})$$
$$M_7 = (A_{12} - A_{22}) \times (B_{21} + B_{22})$$

Strassen's Algorithm

- Problem: Determine the product of two $n \times n$ matrices where n is a power of 2.
- Inputs: An integer n that is a power of 2, and two $n \times n$ matrices A and B.
- Outputs: the product C of A and B.

```
void strassen (int n, n*n_matrix A, n*n_matrix B, n*n_matrix& C) {
    if (n <= threshold)
        Compute C = A * B;
    else {
        partition A into four submatrices $A_{11}$, $A_{12}$, $A_{21}$, $A_{22}$;
        partition B into four submatrices  $B_{11}$, $B_{12}$, $B_{21}$, $B_{22}$;
        Compute C = A * B using strassen's method;
        // example recursive call:
         //strassen(n/2, $A_{11}+A_{12}$, $B_{11}+B_{22}$,$M_1$)
    }
}
```

Every-Case Time complexity Analysis of Number of Multiplications (Strassen)

- Basic operations: one elementary multiplcation.
- Input size: n, the number of rows and columns.

$$T(n) = 7T(\tfrac{n}{2}) \quad n > 1,\ n = 2^k (k \geq 1)$$

$$T(1) = 1$$

$$\begin{aligned} T(n) &= 7 \times 7 \times \cdots \times 7 \\ &= 7^{\mathrm{k}} \\ &= 7^{\lg \mathrm{n}} \\ &= \mathrm{n}^{\lg 7} \\ &= \mathrm{n}^{2.81} \\ &\in \Theta(\mathrm{n}^{2.81}) \end{aligned}$$

Every-Case Time complexity Analysis of Number of Additions/Subs (Strassen)

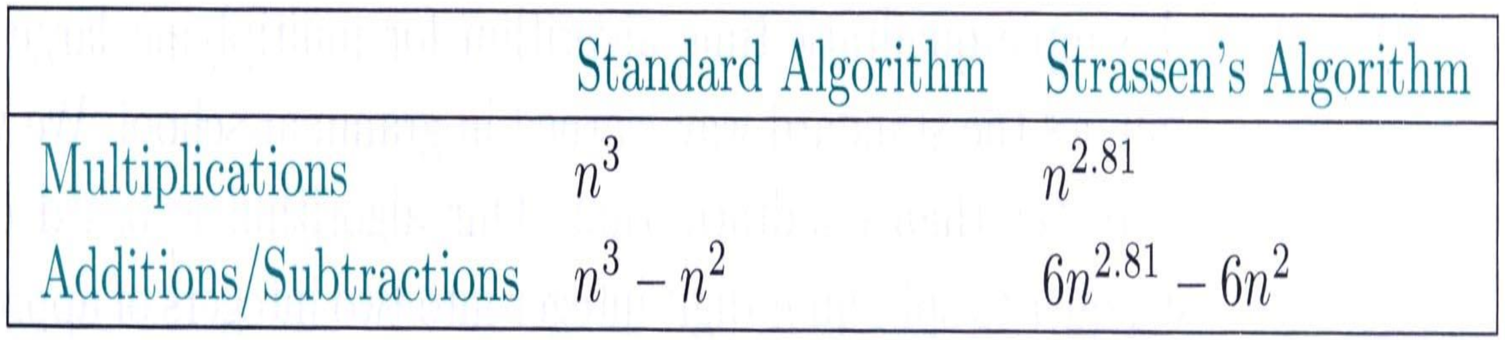
- Basic operations: one elementary addition or subtraction.
- Input size: n, the number of rows and columns.

$$T(n) = 7T(\tfrac{n}{2}) + 18(\tfrac{n}{2})^2 \quad n > 1,\ n = 2^k (k \geq 1)$$

$$T(1) = 0$$

$$T(n) = \Theta(n^{\lg_2 7}) = \Theta(n^{2.81})$$

Table 2.3 A comparison of two algorithms that multiply *n* x *n* matrices



	Standard Algorithm	Strassen's Algorithm
Multiplications	n^3	$n^{2.81}$
Additions/Subtractions	$n^3 - n^2$	$6n^{2.81} - 6n^2$

Comments: Strassen's Algorithm

- Coppersmith and Winograd (1987)
  - $T(n) = O(n^{2.38})$

- However, the constant is so large that Strassen's algorithm is usually more efficient unless the size of n is excessively large.
- No $\Theta(n^2)$ algorithm has been designed.
- Nobody proved that it's impossible!