Chapter 2.

Divide-and-Conquer

Foundations of Algorithms, 5^{th} Ed.

Richard E. Neapolitan

Contents

2.1 Binary Search

- The **Divide-and-Conquer** Approach
  - *divides* an instance of a problem into *two or more smaller instances*.
    - The divided smaller instances are also instances of the problem.
    - If they are still too large to be solved readily,
      - they can be divided into still smaller instances.
    - If solutions to them can be obtained readily,
      - these smaller solutions can be *combined* into the original solution.
  - It is a ***top-down approach***, that is,
    - the solution to a *top-level instance* of a problem is obtained
    - by *going down* and *obtaining solutions* to smaller instances.

2.1 Binary Search

- The steps of Binary Search:
  - If x equals the middle item, then quit. Otherwise:

1. ***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* than the middle item, choose the *right* subarray.
2. ***Conquer*** (solve) the subarray
   - by determining whether x is in that subarray.
   - Unless the subarray is sufficiently small, use *recursion* to do this.
3. *Obtain* the solution to the array from the solution to the subarray.

$x = 18$

$S =$

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

Choose left subarray because $x < 25$

Compare x with 25

10	12	**13**	14	18	20

Compare x with 13

Choose right subarray because $x > 13$

14	**18**	20

Compare x with 18 $\longrightarrow$ Determine that x is present because $x = 18$

ALGORITHM 2.1: Binary Search (Recursive)

```
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);
    }
}
```

2.1 Binary Search

- Implementing the Recursive Binary Search:
  - Note that n, S, and x *are not parameters* to the function *location*.
    - Only the variables *whose values can change in the recursive calls*
      - are made parameters to recursive routines.
    - Hence, define n, S, and x as *global variables*.
  - Then, our *top-level call* to the function *location* and the output would be:

```
cin >> n;
for (int i = 1; i <= n; i++)
    cin >> S[i];
cin >> x;
int loc = location(1, n);
cout << loc << endl;
```

```
[Input]
13
10 12 13 14 18 20 25 27 30 35 40 45 47
18
[Output]
5
```

2.1 Binary Search

- Time Complexity Analysis (Worst-Case)
  - Basic Operation: the *comparison* of x with $S[mid]$.
  - Input Size: n, the *number of items* in the array.
  - Note that the worst-case can occur
    - when x is larger than all items in the list.
  - Assume that n is a power of 2.
    - If $n = 1$, then $W(n) = W(1) = 1$.
    - If $n > 1$, then $W(n) = \underbrace{W\left(\frac{n}{2}\right)}_{\text{Comparisons in recursive call}} + \underbrace{1}_{\text{Comparison at top level}}$

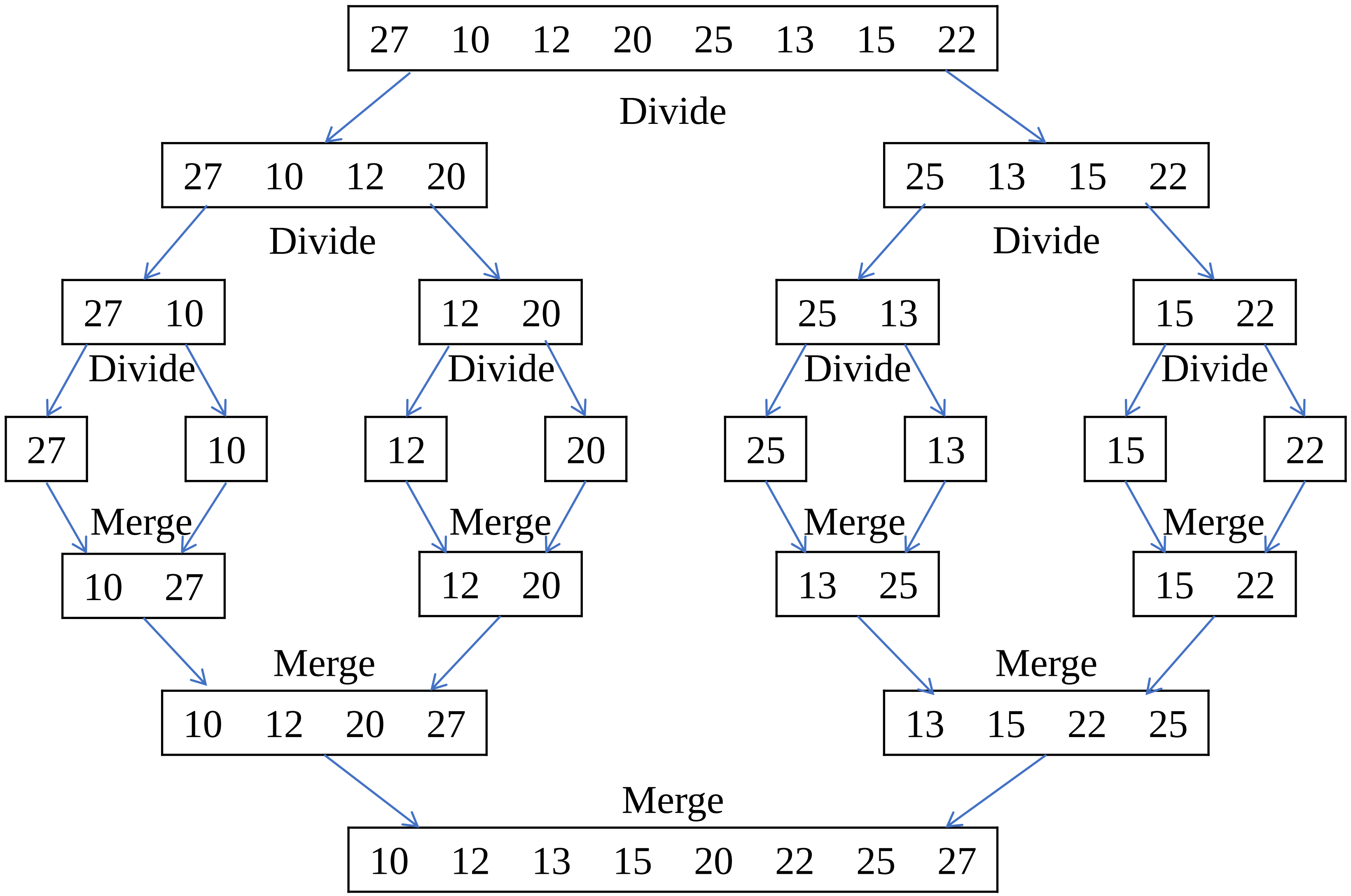
- Time Complexity Analysis (Worst-Case)
  - The recurrence equation:
    - $W(1) = 1$, for $n = 1$,
    - $W(n) = (n/2) + 1$, for $n > 1$ and n is a power of 2.
  - This recurrence is solved to:
    - $W(n) = \lg n + 1 \in \Theta(\lg n)$. (Refer to Example B.1 in Appendix B)
  - If n is not restricted to being a power of 2, then
    - $W(n) = \lfloor \lg n \rfloor + 1 \in \Theta(\lg n)$. (Refer to Exercise 2.1.4)

2.2 Mergesort

- The steps of Mergesort
  1. ***Divide*** the array into two subarrays each with $n/2$ items.
  2. ***Conquer*** (*solve*) each subarray by sorting it.
     - Unless the array is sufficiently small, use *recursion* to do this.
  3. ***Combine*** the solutions to the subarrays
     - by *merging* them into a single sorted array.

- *Two-way merging*
  - combines *two sorted arrays* into *one sorted array*.

2.2 Mergesort



2.2 Mergesort

ALGORITHM 2.2: Mergesort

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

ALGORITHM 2.3: Merge

```
void merge(int h, int m, const keytype U[], const keytype V[], keytype S[]) {
    int 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];
        memcpy(&S[k], &V[j], (m - j + 1) * sizeof(keytype));
    else
        // copy U[i] through U[h] to S[k] through S[h+m];
        memcpy(&S[k], &U[i], (m - i + 1) * sizeof(keytype));
}
```

2.2 Mergesort

- Merging two arrays U and V into one array S.

U	V	S
==10== 12 20 27	==13== 15 22 25	10
==12== 20 27	==13== 15 22 25	10 12
==20== 27	==13== 15 22 25	10 12 13
==20== 27	==15== 22 25	10 12 13 15
==20== 27	==22== 25	10 12 13 15 20
==27==	==22== 25	10 12 13 15 20 22
==27==	==25==	10 12 13 15 20 22 25
27		10 12 13 15 20 22 25 27

2.2 Mergesort

- Time Complexity of *Merge* (Worst-Case)
  - 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.
  - The *worst-case* occurs when the while-loop is exited,
    - one of two indices (i) has reached its exit point ($h + 1$),
    - whereas the other index (j) has reached m (1 less than its exit point).
  - Therefore,
    - $W(h, m) = h + m - 1$.

2.2 Mergesort

- Time Complexity of Mergesort (Worst-Case)
  - Basic Operation: the *comparison* that takes place in $merge$.
  - Input Size: n, the *number of items* in the array S.
  - The total number of comparisons is the sum of
    - the number of comparison in the recursive call to $mergesort$.

$$W(n) = \underbrace{W(h)}_{\text{Time to sort } U} + \underbrace{W(m)}_{\text{Time to sort } V} + \underbrace{h + m - 1}_{\text{Time to merge}}$$

- Time Complexity of Mergesort (Worst-Case)
  - In the case where n is a power of 2.
    - Establish the recurrence relation:
      - $h = \lfloor n/2 \rfloor = n/2$, $m = n - h = n/2$, $h + m = n$.
      - $W(1) = 0$, for $n = 1$,
      - $W(n) = 2W(n/2) + n - 1$, for $n > 1$, n is a power of 2.
    - Therefore,
      - $W(n) = n \lg n - (n - 1) \in \Theta(n \lg n)$ (Example B.19 in Appendix B)
  - In the case where n is not a power of 2.
    - $W(n) = W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + n - 1$
    - $W(n) \in \Theta(n \lg n)$ by Theorem B.4 (Example B.25 in Appendix B.4)

2.2 Mergesort

- How about the Space Complexity?
  - An ***in-place sort*** is a sorting algorithm that
    - does not use any extra space beyond that needed to store the input.
  - Algorithm 2.2 is *not an in-place sort*,
    - because it uses extra arrays U and V besides the input array S.
  - The total number of extra array items created is about
    - $n(1 + \frac{1}{2} + \frac{1}{4} + \cdots) = 2n$
  - It is *possible* to *reduce* the *amount of extra space*
    - to *only one array* containing n items.

2.2 Mergesort

ALGORITHM 2.4: Mergesort 2

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

ALGORITHM 2.5: Merge 2

```
void merge2(int low, int mid, int high) {
    keytype U[high - low + 1];
    int i = low, j = mid + 1, k = 0;
    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)
        memcpy(&U[k], &S[j], (high - j + 1) * sizeof(keytype));
    else
        memcpy(&U[k], &S[i], (mid - i + 1) * sizeof(keytype));
    memcpy(&S[low], &U[0], (high - low + 1) * sizeof(keytype));
}
```

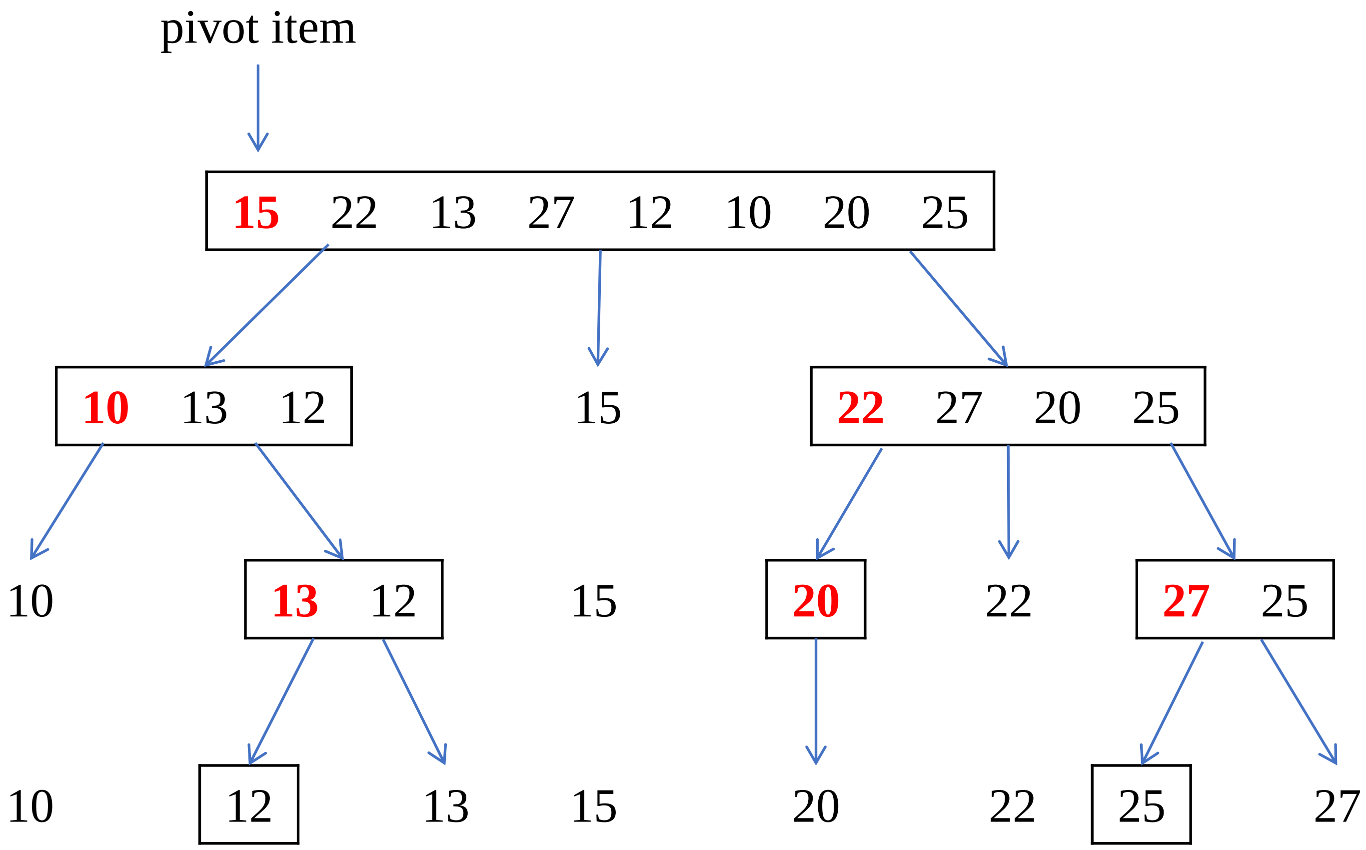
2.3 The Divide-and-Conquer Approach

- The *Design Strategy* of the Divide-and-Conquer:
  1. ***Divide*** an instance of a problem into one or more smaller instances.
  2. ***Conquer*** (*solve*) each of the smaller instances.
     - Unless a smaller instance is sufficiently small, use *recursion* to do this.
  3. *If necessary*, ***combine*** the solutions to the smaller instances
     - to obtain the solution to the original instance.

2.4 Quicksort (Partition Exchange Sort)

- Quicksort
  - is an *in-place* sorting algorithm developed by Hoare (1962).
  - is similar to Mergesort in that
    - it divides the array into *two partitions*
      - and then sorting each partition *recursively*.
  - However, the array is *partitioned*
    - by placing all items *smaller* than some ***pivot item*** *before* that item
      - and all items *larger* than the *pivot item after* it.
    - the *pivot item* can be *any* item,
      - *for convenience*, we will simply make it the *first one*.

2.4 Quicksort (Partition Exchange Sort)



2.4 Quicksort (Partition Exchange Sort)

ALGORITHM 2.6: Quicksort

```
void quicksort(int low, int high) {
    int pivotpoint;

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

2.4 Quicksort (Partition Exchange Sort)

Algorithm 2.7: Partition

```
void partition(int low, int high, int& pivotpoint) {
    int i, j;
    keytype pivotitem;

    pivotitem = S[low];
    j = low;
    for (i = low + 1; i <= high; i++)
        if (S[i] < pivotitem) {
            j++;
            // exchange S[i] and S[j]
            {keytype t = S[i]; S[i] = S[j]; S[j] = t;}
        }
    pivotpoint = j;
    // exchange S[low] and S[pivotpoint]
    {keytype t = S[low]; S[low] = S[pivotpoint]; S[pivotpoint] = t;}
}
```

2.4 Quicksort (Partition Exchange Sort)

$S[1]$	$S[2]$	$S[3]$	$S[4]$	$S[5]$	$S[6]$	$S[7]$	$S[8]$
15	22	13	27	12	10	20	25

15	**22**	13	27	12	10	20	25
j	i						

15	22	**13**	27	12	10	20	25
	j	i					

15	13	22	**27**	12	10	20	25
	j		i				

15	13	22	27	**12**	10	20	25
		j		i			

15	13	12	27	22	**10**	20	25
			j		i		

15	13	12	10	22	27	**20**	25
			j			i	

15	13	12	10	22	27	20	**25**
			j				i

10	13	12	15	22	27	20	25	
			j					i
			pivotpoint					

- Time Complexity of *Partition* (Every-Case)
  - Basic Operation: the *comparison* of $S[i]$ with $pivotitem$.
  - Input Size: $n = high - low + 1$, the *number of items* in the subarray.
  - Since every item except the first is compared,
    - $T(n) = n - 1$.

- Time Complexity of *Quicksort* (Worst-Case)
  - Basic Operation: the *comparison* of $S[i]$ with *pivotitem* in partition.
  - Input Size: n, the *number of items* in the array S.
  - Note that the *worst-case* occurs
    - when the array is *already sorted* in non-decreasing order.
  - If the array is already sorted,
    - *no items are less than* the *first item* (*pivot item*) in the array.
  - Therefore,

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

2.4 Quicksort (Partition Exchange Sort)

- Time Complexity of Quicksort (Worst-Case)
  - recurrence equation:
    - $T(0) = 1$, for $n = 0$,
    - $T(n) \leq \frac{n(n-1)}{2}$, for $n > 0$.
  - the worst-case time complexity is:
    - $W(n) = \frac{n(n-1)}{2} \in \Theta(n^2)$. (Example B.16 in Appendix B)

2.4 Quicksort (Partition Exchange Sort)

- Time Complexity of *Quicksort* (Average-Case)
  - Now assume that the value of *pivotpoint* returned by *partition*
    - is *equally likely* to be *any* of the numbers from 1 through n.
  - In this case, the average-case time complexity is given:

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

Probability that pivotpoint is p

Time to partition

  - The approximate solution to this recurrence is given:
    - $A(n) \approx (n+1) 2 \ln n = (n+1) 2 \ln 2 (\lg n) \approx 1.38(n+1) \lg n \in \Theta(n \lg n)$

- The Analysis of *Recursive Algorithms*:
  - is not as straightforward as it is for *iterative algorithms*.
  - However, it is not difficult to represent
    - the time complexity of a recursive algorithm
    - by a *recurrence equation*.
  - Fortunately, there exist a simple method
    - to solve the recurrence equations with a certain type.
    - called as ***The Master Theorem.***

- **Theorem B.5 (Master Theorem)**
  - Suppose that a complexity function $T(n)$ satisfies:
    - $T(n) = aT(\frac{n}{b}) + cn^k$, for $n > 1$, n is a power of b,
    - $T(1) = d$, for $n = 1$.
      - where $b \geq 2$ and $k \geq 0$ are constant integers,
      - and a, c, and d are constants such that $a > 0$, $c > 0$, and $d \geq 0$.
  - Then,
    - $T(n) \in \Theta(n^k)$, if $a < b^k$.
    - $T(n) \in \Theta(n^k \lg n)$, if $a = b^k$.
    - $T(n) \in \Theta(n^{\log_b a})$, if $a > b^k$.

- Examples of Applying the Master Theorem:
  - Example B.26:
    - $T(n) = 8T(n/4) + 5n^2$, for $n > 1$, n is a power of 4.
    - $T(1) = 3$
    - Then, $T(n) \in \Theta(n^2)$, since $a = 8 < b^k = 4^2$.
  - Example B.27:
    - $T(n) = 9T(n/3) + 5n^1$, for $n > 1$, n is a power of 3.
    - $T(1) = 7$
    - Then, $T(n) \in \Theta\left(n^{\log_3 9}\right) = \Theta(n^2)$, since $a = 9 > b^k = 3^1$.

- Examples of Applying the Master Theorem:
  - Example B.28:
    - $T(n) = 8T(n/2) + 5n^3$, for $n > 64$, n is a power of 2.
    - $T(64) = 200$
    - Then, $T(n) \in \Theta(n^3 \lg n)$, since $a = 8 = b^k = 2^3$.
  - The Analysis of the Algorithm 2.2 (*Mergesort*)
    - $W(n) = 2W(n/2) + n - 1$, for $n > 1$, n is a power of 2.
    - $W(1) = 0$
    - Then, $W(n) \in \Theta(n \lg n)$, since $a = 2 = b^k = 2^1$.

2.5 Strassen's Matrix Multiplication Algorithm

- Matrix Multiplication Algorithm
  - Recall that Algorithm 1.4 multiplies two matrices
    - strictly according *to the definition of matrix multiplication*.
    - time complexity: $T(n) = n^3 \in \Theta(n^3)$.
  - Is it possible to design an efficient algorithm
    - whose time complexity is better than $\Theta(n^3)$?
  - Strassen, in 1969, published an algorithm
    - whose *time complexity* is *better than cubic*
    - in terms of both *multiplication* and *additions/subtractions*.

2.5 Strassen's Matrix Multiplication Algorithm

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

8 multiplications

4 additions

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

7 multiplications

18 additions/subtractions

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

2.5 Strassen's Matrix Multiplication Algorithm

- Pertaining the Strassen's Method to *Larger Matrices*
  - that are each ***divided*** into *four submatrices*.

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

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$
$$M_2 = (A_{21} + A_{22})B_{11}$$
$$\dots$$

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

2.5 Strassen's Matrix Multiplication Algorithm

$$\left[\begin{array}{c:c} C_{11} & C_{12} \\ \hdashline C_{21} & C_{22} \end{array}\right] = \left[\begin{array}{cc:cc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ \hdashline 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{array}\right] \times \left[\begin{array}{cc:cc} 8 & 9 & 1 & 2 \\ 3 & 4 & 5 & 6 \\ \hdashline 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \end{array}\right]$$

$$M_1 = (A_{11} + A_{22})(B_{11} + B_{22}) = \begin{bmatrix} 3 & 5 \\ 11 & 13 \end{bmatrix} \times \begin{bmatrix} 17 & 10 \\ 7 & 9 \end{bmatrix} = \begin{bmatrix} 86 & 75 \\ 278 & 227 \end{bmatrix}$$

$$M_2 =$$

$$M_3 =$$

$$M_4 =$$

$$M_5 =$$

$$M_6 =$$

$$M_7 =$$

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} \quad \end{bmatrix}$$

2.5 Strassen's Matrix Multiplication Algorithm

ALGORITHM 2.8: Strassen (pseudo-code)

```
void strassen(int n, int A[][MAX], int B[][MAX], int C[][MAX]) {
    if (n <= threshold) {
        // compute C = A * 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 * B using Strassen's method;
            // example recursive call:
            // strassen(n/2, A11 + A22, B11 + B22, M1);
    }
}
```

2.5 Strassen's Matrix Multiplication Algorithm

```
#define MAX 32
const int threshold = 2;

void madd(int n, int A[][MAX], int B[][MAX], int C[][MAX]);
void msub(int n, int A[][MAX], int B[][MAX], int C[][MAX]);
void mmult(int n, int A[][MAX], int B[][MAX], int C[][MAX]);
void partition(int n, int A[][MAX],
               int A11[][MAX], int A12[][MAX], int A21[][MAX], int A22[][MAX]);
void combine(int n, int A[][MAX],
               int A11[][MAX], int A12[][MAX], int A21[][MAX], int A22[][MAX]);
void strassen(int n, int A[][MAX], int B[][MAX], int C[][MAX]);
```

2.5 Strassen's Matrix Multiplication Algorithm

```
void partition(int n, int A[][MAX],
               int A11[][MAX], int A12[][MAX], int A21[][MAX], int A22[][MAX])
{
    int m = n / 2;
    for (int i = 0; i < m; i++)
        for (int j = 0; j < m; j++) {
            A11[i][j] = A[i][j];
            A12[i][j] = A[i][j + m];
            A21[i][j] = A[i + m][j];
            A22[i][j] = A[i + m][j + m];
        }
}
```

```
void strassen(int n, int A[][MAX], int B[][MAX], int C[][MAX]) {
    int A11[MAX][MAX], A12[MAX][MAX], A21[MAX][MAX], A22[MAX][MAX];
    int B11[MAX][MAX], B12[MAX][MAX], B21[MAX][MAX], B22[MAX][MAX];
    int C11[MAX][MAX], C12[MAX][MAX], C21[MAX][MAX], C22[MAX][MAX];
    int M1[MAX][MAX], M2[MAX][MAX], M3[MAX][MAX], M4[MAX][MAX],
        M5[MAX][MAX], M6[MAX][MAX], M7[MAX][MAX];
    int L[MAX][MAX], R[MAX][MAX];

    if (n <= threshold) {
        mmult(n, A, B, C);
    }
    else {
        int m = n / 2;

        partition(n, A, A11, A12, A21, A22);
        partition(n, B, B11, B12, B21, B22);

        // Implement Strassen's method here.
    }
}
```

2.5 Strassen's Matrix Multiplication Algorithm

$$m_1 = (a_{11} + a_{22})(b_{11} + b_{22})$$

```
madd(m, A11, A22, L);
madd(m, B11, B22, R);
strassen(m, L, R, M1);
```

$$m_2 = (a_{21} + a_{22})b_{11}$$

```
madd(m, A21, A22, L);
strassen(m, L, B11, M2);
```

$$m_3 = a_{11}(b_{12} - b_{22})$$

```
msub(m, B12, B22, R);
strassen(m, A11, R, M3);
```

$$m_4 = a_{22}(b_{21} - b_{11})$$

```
msub(m, B21, B11, R);
strassen(m, A22, R, M4);
```

$$m_5 = (a_{11} + a_{12})b_{22}$$

```
madd(m, A11, A12, L);
strassen(m, L, B22, M5);
```

… …

```
// Calculate M6, M7 on your own
```

2.5 Strassen's Matrix Multiplication Algorithm

$$C_{11} = m_1 + m_4 - m_5 + m_7$$

```
madd(m, M1, M4, L);
msub(m, L, M5, R);
madd(m, R, M7, C11);
```

$$C_{12} = m_3 + m_5$$

```
madd(m, M3, M5, C12);
```

$$C_{21} = m_2 + m_4$$

```
madd(m, M2, M4, C21);
```

$$C_{22} = m_1 + m_3 - m_2 + m_6$$

```
madd(m, M1, M3, L);
msub(m, L, M2, R);
madd(m, R, M6, C22);
```

$$C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

```
combine(n, C, C11, C12, C21, C22);
```

2.5 Strassen's Matrix Multiplication Algorithm

- Time Complexity of *Strassen's* (*multiplications*)
  - Basic Operation: one *elementary multiplication*.
  - Input Size: n, the *number of rows and columns* in the matrices.
  - For simplicity,
    - we keep dividing until $n = 1$ ($threshold = 1$).
  - Then, we can establish the recurrence:
    - $T(n) = 7T(n/2)$, for $n > 1$, n is a power of 2.
    - $T(1) = 1$.

- Time Complexity of *Strassen's* (*multiplications*)
  - The recurrence is solved in Example B.2 in Appendix B:
    - $T(n) = n^{\lg 7} \approx n^{2.81} \in \Theta(n^{2.81})$.

$$\left.\begin{aligned} T(n) &= 7 \times T\left(\frac{n}{2}\right) \\ &= 7^2 \times T\left(\frac{n}{2^2}\right) \\ &= \cdots \\ &= 7^k \times T\left(\frac{n}{2^k}\right) \\ &= 7^k \times T(1) \\ &= 7^k \end{aligned}\right\} k = \lg n$$

$$\begin{aligned} T(n) &= 7^{\lg n} \\ &= n^{\lg 7} \\ &\approx n^{2.81} \end{aligned}$$

$$T(n) \in \Theta(n^{2.81})$$

  - We can also apply the Master Theorem.

2.5 Strassen's Matrix Multiplication Algorithm

- Time Complexity of *Strassen's* (*additions/subtractions*)
  - Basic Operation: one *elementary* *addition* or *subtraction*.
  - Input Size: n, the *number of rows and columns* in the matrices.
  - Again, for simplicity, we keep dividing until $n = 1$.
  - Then, we can establish the recurrence:
    - $T(n) = 7T\left(\frac{n}{2}\right) + 18(\frac{n}{2})^2$, for $n > 1$, n is a power of 2.
    - $T(1) = 0$.
  - The recurrence is solved in Example B.20 in Appendix B:
    - $T(n) = 6n^{\lg 7} - 6n^2 \in \Theta(n^{2.81})$
  - We can also apply the Master Theorem.

2.5 Strassen's Matrix Multiplication Algorithm

- Comparing two algorithms:

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

- What happen if n is not a power of 2?
  - Simply, fill 0s to the matrices to make the dimension a power of 2.

2.5 Strassen's Matrix Multiplication Algorithm

- *How fast* can we *multiply two matrices*?
  - There are some variants of Strassen's algorithm.
    - Some of them has more efficient complexity, to say, $\Theta(n^{2.38})$.
  - It is *provable* that the complexity requires *at least* $\Omega(n^2)$.
    - This is a *lower bound* of matrix multiplication *problem*.
  - Is it *possible* to design an efficient algorithm with $\Theta(n^2)$?
    - *No one* has ever *developed* an algorithm for it.
    - *No one* has ever *proved* that it is *not possible*.

2.6 Arithmetic with Large Integers

- Representation of Large Integers
  - Suppose that we need to do arithmetic operations on large integers
    - whose size *exceeds* the computer's *hardware capability*.
  - A straightforward way to represent a large integer is
    - to use an array of integers,
    - in which *each array slot* stores only *one digit*.

5	3	4	1	2	7
$S[5]$	$S[4]$	$S[3]$	$S[2]$	$S[1]$	$S[0]$

2.6 Arithmetic with Large Integers

- Data Type and Linear-Time Operations:
  - To represent both *positive* and *negative* integers
    - we need *only* reserve the *high-order array slot* for the *sign*.
      - 0 for positive, 1 for negative.

```
#define MAX 256
#define BASE 10

typedef struct large_integer {
    int sign;
    int len;
    int digits[MAX]; // stored in reverse order.
} large_integer;
```

2.6 Arithmetic with Large Integers

- Data Type and Linear-Time Operations:
  - Write linear-time algorithms for
    - addition & subtraction.
    - powered by exponent: $u \times 10^m$
    - divided by exponent: $u \text{ divide } 10^m$
      - returns the quotient in integer division.
    - remainder by exponent: $u \text{ rem } 10^m$
      - return the remainder.

```
void create_largeint(large_integer &u, char *str);
int compare_by_abs(large_integer u, large_integer v);
void lsum(large_integer u, large_integer v, large_integer &r);
void ldiff(large_integer u, large_integer v, large_integer &r);
void ladd(large_integer &u, large_integer &v, large_integer &r);
void lsub(large_integer u, large_integer v, large_integer &r);
void pow_by_exp(large_integer u, int m, large_integer &v);
void div_by_exp(large_integer u, int m, large_integer &v);
void rem_by_exp(large_integer u, int m, large_integer &v);
```

- Addition & Subtraction

carry	0	0	1	1	0	0	0
		5	6	7	8	3	2
+	9	4	2	3	7	2	3
	9	9	9	1	5	5	5

borrow	-1	-1	-1	-1	-1	0	0
	9	4	2	3	7	2	3
−		5	6	7	8	3	2
	8	8	5	6	8	9	1

```
void lsum(large_integer u, large_integer v, large_integer &r) {
    int k = 0;
    while (k < u.len && k < v.len) {
        r.digits[k] = u.digits[k] + v.digits[k];
        k++;
    }
    for (; k < u.len; k++)
        r.digits[k] = u.digits[k];
    for (; k < v.len; k++)
        r.digits[k] = v.digits[k];
    int carry = 0, i = 0;
    for (; i < k; i++) {
        int d = r.digits[i] + carry;
        r.digits[i] = d % BASE;
        carry = d / BASE;
    }
    if (carry > 0)
        r.digits[i++] = carry;
    r.len = i;
}
```

2.6 Arithmetic with Large Integers

```
void ldiff(large_integer u, large_integer v, large_integer &r) {
    int k = 0;
    while (k < u.len && k < v.len) {
        r.digits[k] = u.digits[k] - v.digits[k];
        k++;
    }
    for (; k < u.len; k++)
        r.digits[k] = u.digits[k];
    int borrow = 0, i = 0;
    for (; i < k; i++) {
        int d = r.digits[i] - borrow;
        r.digits[i] = (d >= 0) ? d: d + BASE;
        borrow = (d >= 0) ? 0: 1;
    }
    while (i > 0 && r.digits[i - 1] == 0)
        i--;
    r.len = i;
}
```

```
void ladd(large_integer &u, large_integer &v, large_integer &r) {
    if (u.sign == v.sign) {
        lsum(u, v, r);
        r.sign = u.sign;
    } else {
        switch (compare_by_abs(u, v)) {
            case 1:
                ldiff(u, v, r);
                r.sign = u.sign;
                break;
            case -1:
                ldiff(v, u, r);
                r.sign = v.sign;
                break;
            case 0:
                r.sign = r.len = 0;
                break;
        }
    }
}
```

2.6 Arithmetic with Large Integers

- Operations with Exponents: Power, Divide, and Remainder

$u = 567832, m = 3$

$u \times 10^m$ $\quad u = 567832000$

$u \text{ divide } 10^m$ $\quad u = 567\cancel{832}$

$u \text{ rem } 10^m$ $\quad u = \cancel{567}832$

2.6 Arithmetic with Large Integers

```
void pow_by_exp(large_integer u, int m, large_integer &v) {
    v.sign = u.sign;
    v.len = u.len + m;
    for (int i = 0; i < m; i++)
        v.digits[i] = 0;
    for (int i = 0; i < u.len; i++)
        v.digits[m + i] = u.digits[i];
}


void div_by_exp(large_integer u, int m, large_integer &v) {
    v.sign = u.sign;
    v.len = u.len - m;
    for (int i = 0; i < u.len; i++)
        v.digits[i] = u.digits[m + i];
}
```

2.6 Arithmetic with Large Integers

- *Multiplication* of *Large Integers*:
  - A simple algorithm for multiplying large integers
    - has a quadratic time complexity: $\Theta(n^2)$.

		1	2	3
$\times$			4	5
		5	10	15
$+$	4	8	12	
	4	13	22	15
	5	5	3	5

2.6 Arithmetic with Large Integers

```
void lmult(large_integer u, large_integer v, large_integer &r) {
    r.sign = (u.sign == v.sign) ? u.sign: 1;
    r.len = u.len + v.len - 1;
    for (int i = 0; i < r.len; i++)
        r.digits[i] = 0;
    for (int i = 0; i < v.len; i++)
        for (int j = i; j < i + u.len; j++)
            r.digits[j] += v.digits[i] * u.digits[j];
    int carry = 0, i = 0;
    for (; i < r.len; i++) {
        int d = r.digits[i] + carry;
        r.digits[i] = d % BASE;
        carry = d / BASE;
    }
    if (carry > 0)
        r.digits[i++] = carry;
    r.len = i;
}
```

2.6 Arithmetic with Large Integers

- Designing an *Efficient* Multiplication Algorithm:
  - based on using the Divide-and-Conquer approach
    - to *split* an n-digit integer into two integers of *approximately* $n/2$ digits.

$$\underbrace{567{,}832}_{6\text{ digits}} = \underbrace{567}_{3\text{ digits}} \times 10^3 + \underbrace{832}_{3\text{ digits}}$$

$$\underbrace{9{,}423{,}723}_{7\text{ digits}} = \underbrace{9{,}423}_{4\text{ digits}} \times 10^3 + \underbrace{723}_{3\text{ digits}}$$

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

The exponent m of 10 is given by $m = \lfloor n/2 \rfloor$

2.6 Arithmetic with Large Integers

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

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

$$\begin{aligned} uv &= (x \times 10^m + y)(w \times 10^m + z) \\ &= xw \times 10^{2m} + (xz + wy) \times 10^m + yz \end{aligned}$$

$$\begin{aligned} 567{,}832 \times 9{,}423{,}723 &= (567 \times 10^3 + 832)(9423 \times 10^3 + 723) \\ &= 567 \times 9{,}423 \times 10^6 + (567 \times 723 + 9423 \times 832) \times 10^3 + 832 \times 723 \end{aligned}$$

2.6 Arithmetic with Large Integers

ALGORITHM 2.9: Large Integer Multiplication

```
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 10^m; y = u rem 10^m;
        w = v divide 10^m; z = v rem 10^m;
        return prod(x, w)×10^2m + (prod(x, z) + prod(w, y))×10^m + prod(y, z);
    }
}
```

2.6 Arithmetic with Large Integers

```
void prod(large_integer u, large_integer v, large_integer &r) {
    large_integer x, y, w, z, t1, t2, t3, t4, t5, t6, t7, t8;
    int n, m;
    n = (u.len > v.len) ? u.len: v.len;
    if (u.len == 0 && v.len == 0)
        r.sign = r.len = 0;
    else if (n <= threshold)
        lmult(u, v, r);
    else {
        int m = n / 2;
        div_by_exp(u, m, x); rem_by_exp(u, m, y);
        div_by_exp(v, m, w); rem_by_exp(v, m, z);
        prod(x, w, t1); pow_by_exp(t1, 2*m, t2);
        prod(x, z, t3); prod(w, y, t4); ladd(t3, t4, t5); pow_by_exp(t5, m, t6);
        prod(y, z, t7);
        ladd(t2, t6, t8); ladd(t8, t7, r);
    }
}
```

- Time Complexity of Algorithm 2.9 (Worst-Case)
  - Basic Operation: the *manipulation of one decimal digit* in a large integer
    - when *adding*, *subtracting*, or doing *pow*, *div*, and *rem* operations.
  - Input Size: n, the *number of digits* in each of the two integers.
  - The worst-case occurs when
    - both integers have *no digits equal to* 0,
      - because the recursion ends if and only if *threshold* is passed.
  - For simplicity, suppose that n is a power of 2.

2.6 Arithmetic with Large Integers

- Time Complexity of Algorithm 2.9 (Worst-Case)
  - The operations of addition, subtraction, power, divide, and remainder
    - have linear time-complexities in terms of n, because $m = n/2$.
  - We can establish the recurrence equation:
    - $W(n) = 4W(n/2) + cn$, for $n > s$, n is a power of 2.
      - where c is a positive constant.
    - $W(s) = 0$, for $n <= s$.
  - Therefore,
    - $W(n) \in \Theta(n^{\lg 4}) = \Theta(n^2)$. (Example B.25 in Appendix B)
      - We can apply the Master Theorem.

2.6 Arithmetic with Large Integers

- What's happen?
  - Algorithm 2.9 is still quadratic: $\Theta(n^2)$
  - The algorithm does *four multiplications*
    - on integers with *half* as many digits as the original integers.
  - We should *reduce the number of these multiplications*.
    - to obtain an algorithm that is *better than quadratic*.

2.6 Arithmetic with Large Integers

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

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

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

$$r = (x + y)(w + z) = xw + (xz + yw) + yz$$

$$(xz + yw) = r - (xw + yz)$$

$$uv = xw \times 10^{2m} + \big((x + y)(w + z) - (xw + yz)\big) \times 10^m + yz$$

three multiplications

2.6 Arithmetic with Large Integers

ALGORITHM 2.10: Large Integer Multiplication 2

```
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;
    }
}
```

- Time Complexity of Algorithm 2.10 (Worst-Case)
  - If n is a power of 2, then x, y, w, and z all have $n/2$ digits.
    - $\frac{n}{2}$ <= digits in $x + y \leq \frac{n}{2} + 1$.
    - $\frac{n}{2}$ <= digits in $w + z \leq \frac{n}{2} + 1$.

n	x	y	$x + y$	Number of Digits in $x + y$
4	10	10	20	$2 = n/2$
4	99	99	198	$3 = n/2 + 1$
8	1000	1000	2000	$4 = n/2$
8	9999	9999	19,998	$5 = n/2 + 1$

- **Time Complexity of Algorithm 2.10 (Worst-Case)**
  - The input sizes for the given function calls:
    - $prod2(x+y, w+z)$: $\frac{n}{2} \leq$ input size $\leq \frac{n}{2} + 1$.
    - $prod2(x, w)$: input size $= \frac{n}{2}$
    - $prod2(y, z)$: input size $= \frac{n}{2}$
  - Therefore, $W(n)$ satisfies
    - $3W(\frac{n}{2}) + cn \leq W(n) \leq 3W(\frac{n}{2} + 1) + cn$, for $n > s$, n is a power of 2.
    - $W(s) = 0$, for $n \leq s$.

- Time Complexity of Algorithm 2.10 (Worst-Case)
  - Owing to the left inequality in the recurrence and the Master Theorem:
    - $W(n) \in \Omega(n^{\log_2 3})$.
  - We can also show that
    - $W(n) \in O(n^{\log_2 3})$. (Refer to the textbook)
  - Therefore, combining these two results,
    - $W(n) \in \Theta(n^{\log_2 3})$.

2.7 Determining Thresholds

- The Effect of *Threshold* Value
  - Recursion requires
    - a fair amount of overhead in terms of computer time.
  - Consider the problem of sorting *only eight keys*:
    - Which is the faster in terms of the *execution* time?
      - Recursive Mergesort: $\Theta(n \lg n)$ or Exchange Sort: $\Theta(n^2)$.
  - We need to develop a method that *determines for what value of n*
    - it is at least as fast to call an alternative algorithm as it is
      - to divide the instance further.

2.7 Determining Thresholds

- Finding an ***Optimal Threshold***:
  - An *optimal threshold value* of n is
    - an instance size such that for any smaller instance
      - it would be at least as fast to call the other algorithm as
      - it would be to divide the instance further,
    - and for any larger instance size
      - it would be faster to divide the instance again.

2.7 Determining Thresholds

- Example: Mergesort & Exchange Sort
  - Recurrence of Mergesort (worst-case)
    - $W(n) = 2W(n/2) + 32n\ \mu s$, $W(1) = 0\ \mu s$
  - Mergesort takes $W(n) = 32n \lg n\ \mu s$, where Exchange Sort takes $\frac{n(n-1)}{2}\ \mu s$.
  - Solving the inequality $\frac{n(n-1)}{2} < 32n \lg n$, the solution is $n < 591$.
  - Is it optimal to call Exchange Sort when $n < 591$
    - and to call Mergesort otherwise?
  - Note that this analysis is *incorrect*.
  - It only tells us that if we use Mergesort and keep dividing until $n = 1$,
    - then Exchange Sort is better for $n < 591$.

The Optimal Threshold for Mergesort & Exchange Sort:

- Suppose we modify Mergesort so that
  - Exchange Sort is called when $n \leq t$ for some threshold t.
- $W(n) = \begin{cases} \frac{n(n-1)}{2}\ \mu s, & \text{for } n \leq t \\ W\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + W\left(\left\lceil \frac{n}{2} \right\rceil\right) + 32n\ \mu s, & \text{for } n > t \end{cases}$
  - $W\left(\left\lfloor \frac{t}{2} \right\rfloor\right) + W\left(\left\lceil \frac{t}{2} \right\rceil\right) + 32t = \frac{t(t-1)}{2}$
  - Solving this equation, we can obtain $t = 128$. (Refer to the textbook)
- Therefore, we have
  - an *optimal threshold* value of 128.

2.8 When not to Use Divide-and-Conquer

- Avoid the Divide-and-Conquer in the following two cases:
  1. An instance of size n is divided into
     - *two or more instances* each almost size n.
       - It leads to an *exponential-time* algorithm.
  2. An instance of size n is divided into
     - almost n *instances* of size n/c, where c is a constant.
       - It leads to $n^{\Theta(\lg n)}$ algorithm.

- Consider the following problems:
  - nth Fibonacci Term: Algorithm 1.6 (Recursive), 1.7 (Iterative)
  - Towers of Hanoi: *intrinsically* exponential algorithm.

Any Questions?