Chap 2. Divide-and-Conquer

1. Binary Search
2. Mergesort
3. The Divide-and-Conquer Approach
4. Quicksort (Partition Exchange Sort)
5. Strassen's Matrix Multiplication Algorithm
6. When Not to Use Divide-and-Conquer

Strategy of Divide-and-Conquer

- 분할 (*Divide*)
  - 해결하기 쉽도록 문제를 여러 개의 작은 부분으로 나눈다.
- 정복 (*Conquer* – Solve)
  - 나눈 작은 문제를 각각 해결한다.
- 통합 (*Combine* – Obtain the solution)
  - (필요하다면) 해결된 해답을 모은다.

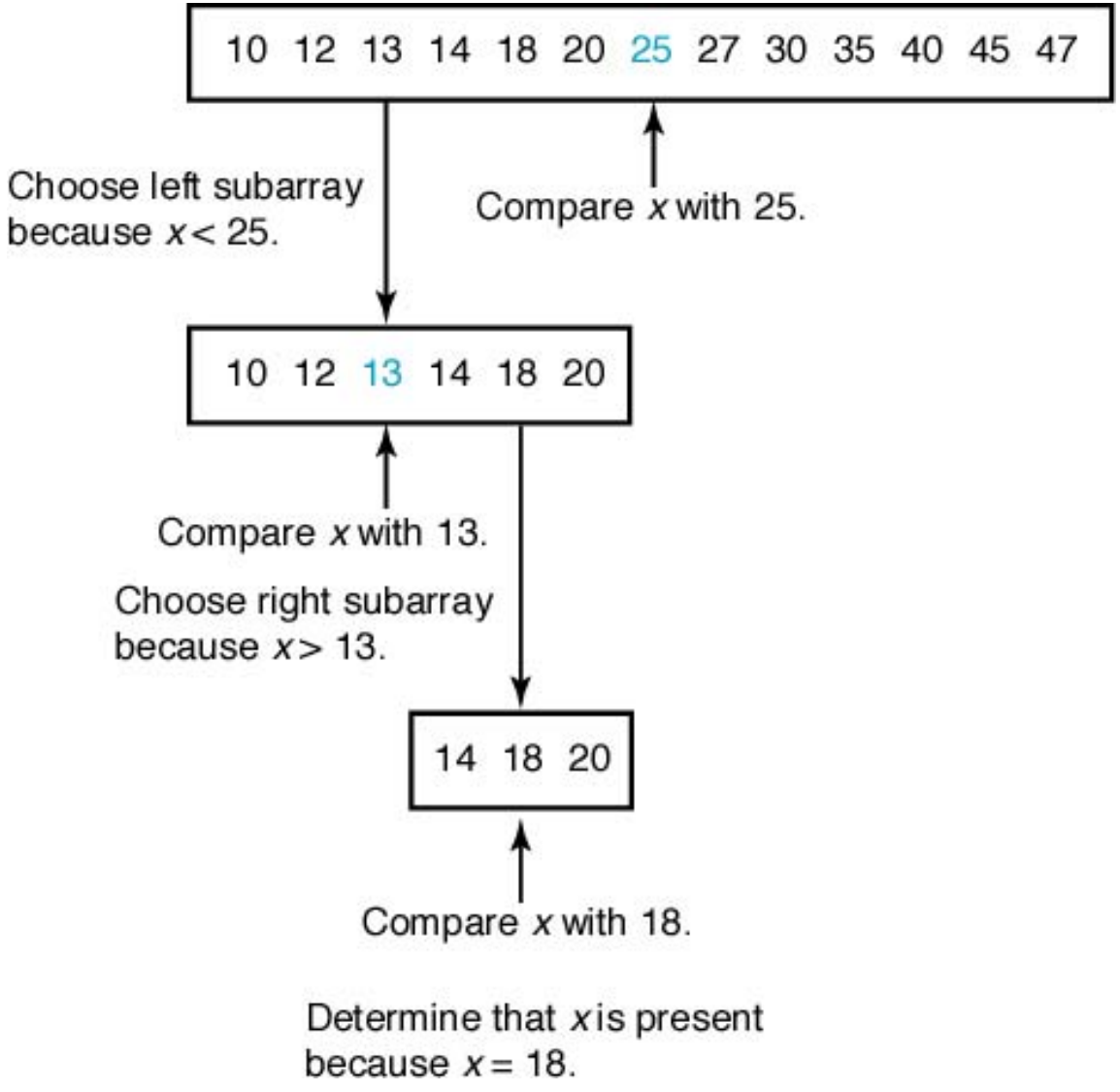
Binary Search

- 재귀 알고리즘

  - 문제: 크기가 *n*인 정렬된 배열 *S*에 *x*가 있는지를 결정하라.
    - 입력: 자연수 *n*, 비내림차순으로 정렬된 배열 *S*[1..n], 찾고자 하는 항목 *x*
    - 출력: *locationout* – *x*가 *S*의 어디에 있는지의 위치.
      만약 *x*가 *S*에 없다면 0

  - 설계전략:
    - *x*가 배열의 중간에 위치하고 있는 항목과 같으면, "빙고", 찾았다!
      그렇지 않으면:
    - *Divide* (분할): 배열을 반으로 나누어서
      *x*가 중앙에 위치한 항목보다 작으면 왼쪽 배열 반쪽을 선택,
      그렇지 않으면 오른쪽 배열 반쪽을 선택한다.
    - *Conquer* (정복): 선택된 반쪽 배열에서 *x*를 찾는다.
    - *Combine* (통합): 필요 없음

Binary Search



Binary Search

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

    if (low > high)
        return 0;                          // 찾지 못했음
    else {
        mid = ⌊(low + high) / 2⌋;          // 정수 나눗셈 (나머지 버림)
        if (x == S[mid])
            return mid;                    //  찾았음
        else if (x < S[mid])
            return location(low, mid-1);     // 왼쪽 반을 선택함
        else
            return location(mid+1, high);  // 오른쪽 반을 선택함
    }
}
…
locationout = location(1, n);              // evoke location
…
```

Notice

- *Why n, S, x are not parameters to function location ?*

  - ☞ They remain unchanged in each recursive call.
  - ☞ A new copy of any variable passed to the routine is made in each recursive call.
  - ☞ So, if a variable's value doesn't change, the copy is unnecessary.
  - ☞ Only the variables, whose values can change in the recursive calls, are made parameters to recursive calls.
  - ☞ It makes the expression of recursive routines cleaner.
  - ☞ An array is automatically passed by address in C/C++.

Worst-Case Time Complexity Analysis

- Binary Search (Recursive)
  - Basic operation: the comparison of *x* with *S[mid]*
  - Input size: n (= *high* - *low* + 1), the number of items in the array

- $n = 2^k$ 인 경우: 다음과 같이 식을 유도할 수 있다.

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

이를 풀면,

$$W(1) = 1$$
$$W(2) = W(1) + 1 = 2$$
$$W(4) = W(2) + 1 = 3$$
$$\cdots$$
$$W(2^k) = k + 1 \;\Rightarrow\; W(n) = \lg n + 1$$

Worst-Case Time Complexity Analysis

– 일반적인 경우: $\text{mid} = \lfloor (n+1)/2 \rfloor$인데, 각 부분배열의 크기는 다음과 같다.

n	왼쪽 부분배열의 크기	mid	오른쪽 부분배열의 크기
짝수	n/2 - 1	1	n/2
홀수	(n-1)/2	1	(n-1)/2

- $W(n) = \lfloor lg\ n \rfloor + 1$ (수학적귀납법 증명)
  - 귀납출발점: $n = 1$이면, 다음이 성립한다.

    $\lfloor lg\ n \rfloor + 1 = \lfloor lg\ 1 \rfloor + 1 = 0 + 1 = 1 = W(1)$
  - 귀납가정: $n > 1$이고, $1 < k < n$인 모든 k에 대해서,

    $W(k) = \lfloor lg\ k \rfloor + 1$ 이 성립한다고 가정

Worst-Case Time Complexity Analysis

- 귀납증명:

  - n이 짝수이면,

$$W(n) = W(n/2) + 1$$
$$= (\lfloor lg\ n/2 \rfloor + 1) + 1$$
$$= (\lfloor lg\ n - 1 \rfloor + 1) + 1$$
$$= \lfloor lg\ n \rfloor + 1$$

  - n이 홀수이면,

$$W(n) = W((n-1)/2) + 1$$
$$= (\lfloor lg\ (n-1)/2 \rfloor + 1) + 1$$
$$= \lfloor lg\ (n-1) \rfloor + 1$$
$$= \lfloor lg\ n \rfloor + 1 \ (n\text{이 홀수이므로})$$

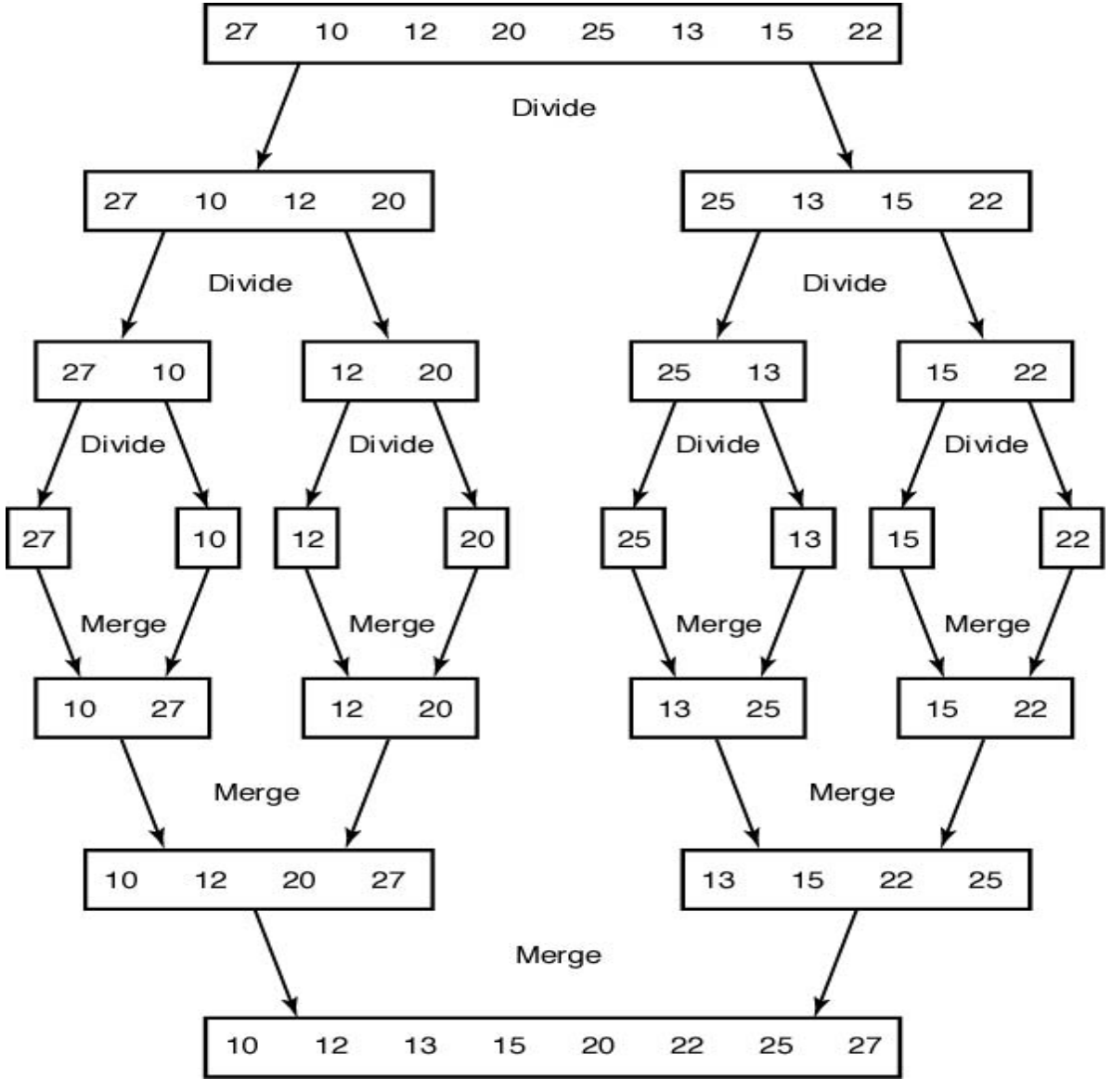
Mergesort

- Mergesort – 합병정렬

- 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

- 보기: 27, 10, 12, 20, 25, 13, 15, 22

Mergesort


27 10 12 20 25 13 15 22
Divide
27 10 12 20
25 13 15 22
Divide
Divide
27 10
12 20
25 13
15 22
Divide
Divide
Divide
Divide
27
10
12
20
25
13
15
22
Merge
Merge
Merge
Merge
10 27
12 20
13 25
15 22
Merge
Merge
10 12 20 27
13 15 22 25
Merge
10 12 13 15 20 22 25 27

Mergesort

- Algorithm

```
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

- Problem: Merge 2 sorted arrays into one sorted array
  - Inputs: (1) positive integers h and m,
    (2) array of sorted keys $U[1..h]$, $V[1..m]$
  - Outputs: an array $S[1..h+m]$ containing keys in U and V in a single sorted array

Merge Algorithm

```
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];
}
```

Merge

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)

- Worst-Case Time Complexity of "Merge" algorithm
  - Basic Analysis: the comparison of *U[i]* with *V[j]*
  - Input Size: *h* and *m*, the number of items in each of the 2 input arrays
  - Analysis:
    - The worst case occurs when the loop is exited, because $i = h+1$ *and* $j = m$.
    - For example, this can occur when the first $m - 1$ items in *V* are placed first in S, followed by all h items in *U*, at which time the loop is exited because $i = h+1$.
    - $T(h,m) = h + m - 1$

Worst-Case Time Complexity (Mergesort)

- Worst-Case Time Complexity of "Mergesort" algorithm
  - Basic Operation: the comparison that takes place in *merge*

    = the comparison of *U[i]* with *V[j]*
  - Input size: *n*, the number of items in the array *S*
  - Analysis:
    - 최악의 경우 수행시간은

$$W(h+m) = W(h) + W(m) + T(h,m) = W(h) + W(m) + (h + m - 1)$$

- $W(h)$: U를 정렬하는데 필요한 횟수
- $W(m)$: V를 정렬하는데 필요한 횟수
- $T(h,m)$: "Merge" 함수에서 필요한 횟수

Worst-Case Time Complexity (Mergesort)

- Case 1: n is a power of 2

  In this case

  - $h = \lfloor n/2 \rfloor = n/2$,
  - $m = n - h = n - n/2 = n/2$,
  - $h + m = n/2 + n/2 = n$

  W(n) becomes

  $W(n) = 2\,W(n/2) + n - 1$ for $n>1$, n a power of 2

  $W(1) = 0$

Worst-Case Time Complexity (Mergesort)

$W(n) = 2\,W(n/2) + n - 1 \quad (\text{for } n>1,\ n = 2^k,\ W(1) = 0)$

$$
\begin{aligned}
W(2^k) &= 2 \cdot W(2^{k-1}) + 2^k - 1 \\
&= 2 \cdot (2 \cdot W(2^{k-2}) + 2^{k-1} - 1) + 2^k - 1 \\
&= 2^2 \cdot W(2^{k-2}) + 2 \cdot 2^k - (2+1) \\
&= 2^3 \cdot W(2^{k-3}) + 3 \cdot 2^k - \textstyle\sum_{(i=0 \sim 2)} 2^i \\
&= \ \ldots\ldots \\
&= 2^k \cdot W(2^0) + k \cdot 2^k - \textstyle\sum_{(i=0 \sim k-1)} 2^i
\end{aligned}
$$

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

Therefore, $W(n) \in \Theta(n \lg n)$

Worst-Case Time Complexity (Mergesort)

- Case 2: n is not a power of 2

$$W(n) = W(\lfloor \tfrac{n}{2} \rfloor) + W(\lceil \tfrac{n}{2} \rceil) + n + 1 \qquad n > 1\text{일 때}$$
$$W(1) = 0$$

이 점화식의 정확한 해를 증명하기는 지저분하다.
그러나, 어떤 k에 대해 $2^k < n < 2^{k+1}$ 을 만족하므로,
$n = 2^k$라고 가정해서 구한 점화식의 해와 같은 카테고리의
시간복잡도를 얻게 된다.
(앞으로 이와 비슷한 점화식의 해를 구할 때,
$n = 2^k$라고 가정해서 구해도 점근적으로는 같은 해를 얻게 된다.)
따라서, $W(n) \in \Theta(n \lg n)$.

Space Complexity (Mergesort)

- ***in-place sort*** (제자리정렬) 알고리즘
  - Doesn't use any extra space beyond that needed to store the input
  - 합병정렬 알고리즘은 제자리정렬 알고리즘이 아니다. 왜냐하면 입력인 배열 S 이외에 U와 V를 추가로 만들어서 사용하기 때문이다.

- 그러면 얼마만큼의 추가적인 저장장소가 필요할까?
  - **재귀호출할 때마다 크기가 S의 반이 되는 U와 V가 추가적으로 필요**

    merge 알고리즘에서는 U와 V가 주소로 전달이 되어 그냥 사용되므로 추가적인 저장장소를 만들지 않는다. 따라서 mergesort를 재귀호출할 때마다 얼마만큼의 추가적인 저장장소가 만들어져야 하는지를 계산해 보면 된다.
    - 처음 S의 크기가 n이면, 추가적으로 필요한 U와 V의 저장장소 크기의 합은 n이 된다.
    - 다음 재귀 호출에서는 $n/2$ 이 추가적으로 필요
    - 결국 총 저장장소의 크기는 $n + n/2 + n/4 + \ldots\ldots = 2n$이 필요하다.

Mergesort2

- Reduce the amount of extra space to only one array containing n items
- Problem: Sort *n* keys in nondecreasing sequence
  - Inputs: Positive Integer *n*, array of keys *S[1..n]*
  - Outputs: the array *S* containing keys in nondecreasing order
- Algorithm:

```
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);
```

Mergesort2

- Mergesort2 에서 별도의 기억공간을 사용하지 않음.

- Merge2
  - Problem: Merge the 2 sorted subarrays of *S* created in Mergesort 2
    - Inputs: (1) indices *low*, *mid*, *high*,
      (2) the subarray of *S[low..high]*,
      where keys in *S[low..mid]* and *S[mid+1..high]* are already sorted in nondecreasing order
    - Outputs: keys in *S[1..high]* in nondecreasing order

공간복잡도가 향상된 합병 (Merge) 알고리즘

```
void merge2(index low, index mid, index high) {
    index i, j, k;
    keytype U[low..high];                    // 합병하는데 필요한 지역 배열
    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];
}
```

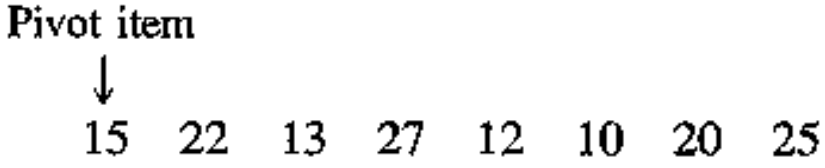
Quicksort (Partition Exchange Sort)

- Quicksort
  - Developed by Hoare (1962)
  - Similar to Mergesort
    - The sort is accomplished by dividing the array into 2 partitions
    - Then sorting each partition recursively
  - But the array is partitioned by a pivot item
    - Divide all items into 2 arrays, smaller/larger than the pivot item
  - Quicksort이란 이름이 오해의 여지가 있음.
    - 사실 절대적으로 가장 빠른 정렬 알고리즘이라고 할 수 없음
    - "Partition Exchange Sort (분할교환정렬)"라고 부르는 게 타당

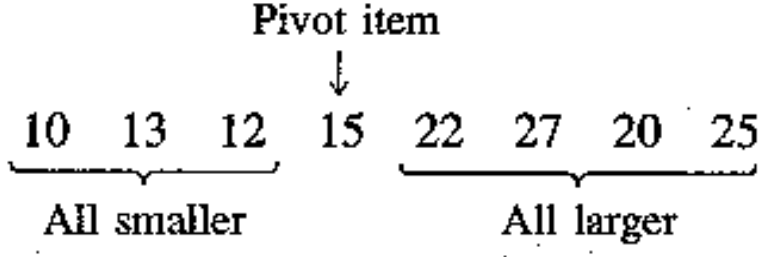
- 보기: 15 22 13 27 12 10 20 25

Quicksort (Partition Exchange Sort)

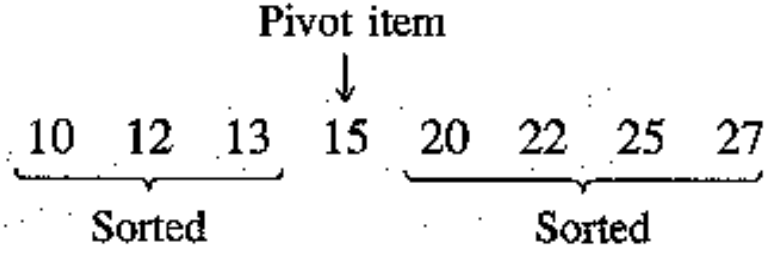
Suppose the array contains these numbers in sequence:



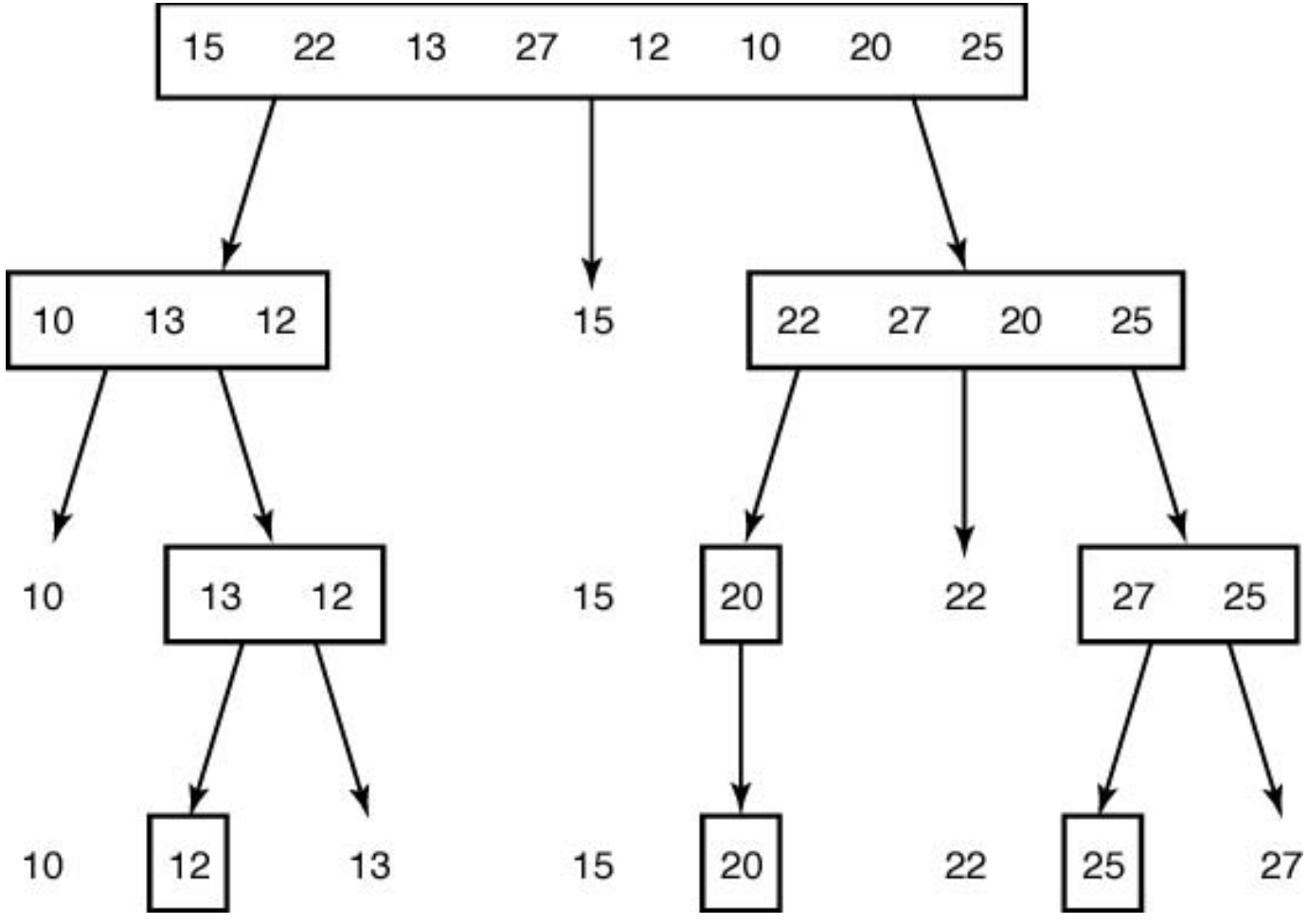
1. Partition the array so that all items smaller than the pivot item are to the left of it and all items larger are to the right:



2. Sort the subarrays:



Quicksort (Partition Exchange Sort)



Quicksort Algorithm

- Problem: Sort n keys in nondecreasing order
  - Inputs: positive integer n, array of keys $S[1..n]$
  - Outputs: the array S containing the keys in nondecreasing order
- Algorithm:

```
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

- Problem: Partition the array S for Quicksort
  - Inputs: (1) 2 indices, *low* & *high*, (2) subarray of *S[low..high]*
  - Outputs: *pivotpoints*, the pivot point for the subarray indexed from *low* to *high*

- Algorithm

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

      pivotitem = S[low];                    // pivotitem=첫번째 항목
      j = low;                               // j=pivotpoint=pivotitem이 위치할 장소
      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];  // pivotitem을 pivotpoint에
}
```

Partition Algorithm

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.

Analysis of Partition Algorithm

- Analysis of 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
  - Analysis: Because every item except the first is compared,

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

Analysis of Quicksort Algorithm

- Analysis of 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*
  - Analysis:
    - The worst case: if the array is already sorted in nondecreasing order
      - In the case, no items are less than the first item in the array,
      - When partition is called at the top level, no items are place to the left, and the value of pivotpoint assigned by partition is 1.
      - In each recursive call, pivotpoint receives the value of low, therefore the array is repeatedly partitioned into an empty subarray on the left & a subarray with one less item on the right

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

    - Because $T(0) = 0$, we have the recurrence

      $T(n) = T(n - 1) + n - 1, \quad \text{for } n > 0,$
      $T(0) = 0$

Analysis of Quicksort Algorithm

이 점화식을 풀면,

$$T(n) = T(n - 1) + n - 1$$
$$T(n - 1) = T(n - 2) + n - 2$$
$$T(n - 2) = T(n - 3) + n - 3$$
$$\cdots$$
$$T(2) = T(1) + 1$$
$$T(1) = T(0) + 0$$
$$T(0) = 0$$

$$T(n) = 1 + 2 + \ldots + (n-1) = n\,(n-1)/2$$

가 되므로, 이미 정렬이 되어 있는 경우 Quicksort 알고리즘의 시간복잡도는 $n\,(n-1)/2$이 된다는 사실을 알았다. 그러면 시간이 더 많이 걸리는 경우가 있을까? 이 경우가 최악이 경우이며, 이보다 더 많은 시간이 걸릴 수가 없다는 사실을 수학적으로 엄밀하게 증명해 보자.

Analysis of Quicksort Algorithm

- Show that $W(n) \le n\,(n-1)/2$ for all n

  Proof: (by using induction)

  - Induction base: for $n = 0$, $W(0) = 0 \le 0\,(0-1)/2$

  - Induction hypothesis: Assume that, for $0 \le k < n$, $W(k) \le k\,(k-1)/2$

  - Induction step: We need to show that $W(n) \le n\,(n-1)/2$

$$W(n) \le W(p-1) + W(n-p) + n - 1 \qquad \text{// pivotpoint 값이 } p\text{인 경우}$$

$$\le \frac{(p-1)(p-2)}{2} + \frac{(n-p)(n-p-1)}{2} + n - 1 \qquad \text{// 귀납가정}$$

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

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

여기서 p가 1 혹은 n일 때 최대값을 가진다.

Analysis of Quicksort Algorithm

따라서 $(p=1$ 일때$)\quad 1-(n+1)+\frac{1}{2}(n^2+n)=\frac{1}{2}n(n-1)$

$(p=n$ 일때$)\quad n^2-(n+1)n+\frac{1}{2}(n^2+n)=\frac{1}{2}n(n-1)$

가 되고,

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

따라서 최악의 경우 시간복잡도는

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

Analysis of Quicksort Algorithm

- Analysis of 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
  - Analysis:
    - 배열 안에 있는 항목이 어떤 특정 순으로 정렬이 되어 있는 경우는 사실 별로 없다. 그러므로 분할 알고리즘이 주는 기준점 값은 1부터 n사이의 어떤 값도 될 수가 있고, **그 확률은 모두 같다고 보아도** 된다.
    - 기준점이 p가 될 확률은 $1/n$이고, 기준점이 p일 때 두 부분배열을 정렬하는데 걸리는 평균시간은 $A(p-1)+A(n-p)$이고, 분할하는데 걸리는 시간은 $n-1$이므로, 평균적인 시간복잡도는 다음과 같이 된다.

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

$$=\frac{2}{n}\sum_{p=1}^{n}A(p-1)+n-1;\ \ (\mathrm{p}-1\text{와 }\mathrm{n}-\mathrm{p}\text{는 대칭이므로.})$$

Analysis of Quicksort Algorithm

양변을 n으로 곱하면, $n\,A(n) = 2\sum_{p=1}^{n} A(p-1) + n(n-1)$ (1)

n대신 n - 1을 대입하면, $(n-1)A(n-1) = 2\sum_{p=1}^{n-1} A(p-1) + (n-1)(n-2)$ (2)

(1)에서 (2)를 빼면, $n\,A(n) - (n-1)A(n-1) = 2\,A(n-1) + 2(n-1)$

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

여기서 $a_n = \frac{A(n)}{n+1}$ 라고 하면, 다음과 같은 점화식을 얻을 수가 있다.

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

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

$a_0 = 0$

Analysis of Quicksort Algorithm

따라서, $a_n = \sum_{i=1}^{n} \frac{2(i-1)}{i(i+1)} = 2\left(\sum_{i=1}^{n} \frac{1}{i+1} - \sum_{i=1}^{n} \frac{1}{i(i+1)}\right)$

여기에서 오른쪽 항은 무시해도 될 만큼 작으므로 무시한다.

$\ln n = \log_e n$ 이고,

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

이므로, 해는 $a_n = A(n)/(n+1) \approx 2 \ln n$.

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

$$\ln n = \frac{\log_2 n}{\log_2 e} = \frac{\log_2 n}{\log_e e / \log_e 2} = \log_e 2 \log_2 n = \ln 2 \lg n$$

Matrix Multiplication

- Simple Matrix Multiplication Algorithm
  - Problem: Determine the product of 2 $n \times n$ matrices
    - Inputs: an integer n, and 2 $n \times n$ matrices A and B
    - Outputs: the product C of A and B
  - Algorithm:

```
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];
          }
}
```

Analysis of Matrix Multiplication

- Every-case Time Complexity Analysis I:
  - 단위연산: 가장 안쪽의 루프에 있는 곱셈하는 연산
  - 입력크기: 행과 열의 수, n
  - 모든 경우 시간복잡도 분석: 총 곱셈의 횟수는

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

- Every-case Time Complexity Analysis II:
  - 단위연산: 가장 안쪽의 루프에 있는 덧셈하는 연산
  - 입력크기: 행과 열의 수, n
  - 모든 경우 시간복잡도 분석: 총 덧셈의 횟수는

$$T(n) = (n-1) \times n \times n = n^3 - n^2 \in \Theta(n^3)$$

Strassen's Matrix Multiplication Algorithm

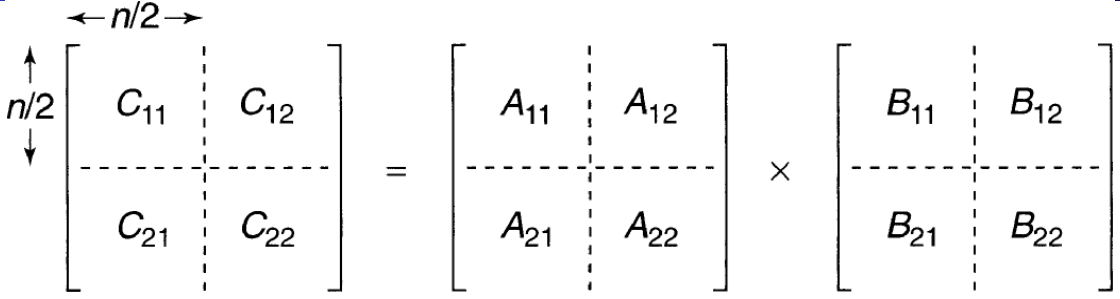
- 문제: n이 2의 거듭제곱이고, 각 행렬을 4개의 부분행렬(submatrix)로 나눈다고 가정
  두 $n \times n$ 행렬 A와 B의 곱 C:

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

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

$$\overset{\leftarrow 2 \rightarrow}{\underset{2}{\updownarrow}} \left[\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right] = \left[\begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ \hline 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \end{array}\right] \times \left[\begin{array}{cc|cc} 8 & 9 & 1 & 2 \\ 3 & 4 & 5 & 6 \\ \hline 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \end{array}\right]$$

Strassen's Matrix Multiplication Algorithm



- Strassen의 방법:

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

여기서

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

- See Figure 2.4 & Example 2.5, page 67-68

Strassen's Matrix Multiplication Algorithm

void strassen (int n, n×n_matrix A, n×n_matrix B, n×n_matrix& C) {
    if (n <= threshold)
        compute $C = A \times B$ using the standard algorithm;
    else {
        partition A into 4 submatrices A_{11}, A_{12}, A_{21}, A_{22};
        partition B into 4 submatrices B_{11}, B_{12}, B_{21}, B_{22};
        partition C into 4 submatrices C_{11}, C_{12}, C_{21}, C_{22};
        strassen$(n/2, A_{11}+A_{22}, B_{11}+B_{22}, M_1)$;
        strassen$(n/2, A_{21}+A_{22}, B_{11}, M_2)$;
        strassen$(n/2, A_{11}, B_{12}-B_{22}, M_3)$;
        strassen$(n/2, A_{22}, B_{21}-B_{11}, M_4)$;
        strassen$(n/2, A_{11}+A_{12}, B_{22}, M_5)$;
        strassen$(n/2, A_{21}-A_{11}, B_{11}+B_{12}, M_6)$;
        strassen$(n/2, A_{12}-A_{22}, B_{21}+B_{22}, M_7)$;
        $C_{11} = M_1+M_4-M_5+M_7$; $C_{12} = M_3+M_5$;
        $C_{21} = M_2+M_4$; $C_{22} = M_1+M_3-M_2+M_6$;
    }
}

Strassen's Matrix Multiplication Algorithm

- Every-case Time Complexity Analysis I (Strassen)
  - Basic operation: one elementary multiplication
  - Input size: *n*, the number of rows and columns in the matrices
  - 모든 경우 시간복잡도 분석:
    - 점화식은 $T(n)=7\ T(\frac{n}{2}),\ n>1$이고, $n=2^k(k\geq 1)$

      $T(1)=1$

      이 식을 전개해 보면,

$$T(n)=7\times 7\times\cdots\times 7\ \ (k\text{번})$$
$$=7^{k}$$
$$=7^{\lg n}$$
$$=n^{\lg 7}\ \ =n^{2.81}\ \ \in\Theta(n^{2.81})$$

Strassen's Matrix Multiplication Algorithm

- Every-case Time Complexity Analysis II (Strassen)
  - Basic operation: one elementary addition or subtraction
  - Input size: n, the number of rows and columns in the matrices
  - 모든 경우 시간복잡도 분석:
    - 점화식은

$$T(n) = 7\ T(\tfrac{n}{2}) + 18(\tfrac{n}{2})^2\ ,\quad n > 1\text{이고}, n = 2^k (k \geq 1)$$
$$T(1) = 0$$

From Example B.20 in Appendix B,

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

Strassen's Matrix Multiplication Algorithm

- 두 개의 행렬을 곱하기 위한 문제에 대해서 시간복잡도가 $\Theta(n^2)$이 되는 알고리즘을 만들어 낸 사람은 아무도 없다.
- 또한, 그러한 알고리즘을 만들 수 없다고 증명한 사람도 아무도 없다.

분할정복을 사용하지 말아야 하는 경우

- 중복된 계산을 반복해 시행하는 경우
  ex) 1장에서 예로 든 Fibonacci 수열 계산 (recursion 이용)

- 크기가 n인 입력이 2개 이상의 조각으로 분할되며,
  분할된 부분들의 크기가 거의 n에 가깝게 되는 경우
  ex) T(n)=T(n-1)+T(n-2) => $T(n)=\alpha(\lambda_1)^n+\beta(\lambda_2)^n$
  ☞ 시간복잡도: 지수(exponential) 시간

- 크기가 n인 입력이 거의 n개의 조각으로 분할되며,
  분할된 부분의 크기가 n/c인 경우, (여기서 c는 상수)
  ex) T(n)= n T(n/c) => $T(n)=\Theta(n^{\log_c n})$
  ☞ 시간복잡도: $n^{\Theta(\lg n)}$