분할정복법(계속)

- 퀵 정렬
- 선택
- Strassen의 행렬 곱셈

2.4 퀵 정렬(quick sort)

- 합병 정렬과는 달리 두 부분 배열로의 분할이 나중에 결합(combine)될 필요가 없다.
- 리스트 (a[1], a[2], ... a[n])에 존재하는 n 개의 원소(키)를 재배치하여 특정 j ($1 \le j \le n$)에 대하여 다음과 같이 세 개의 리스트로 나눈다.

  → 분할(partition)

  P1 = (a[1], a[2], ... a[j-1])

  P2 = (a[j])

  P3 = (a[j+1], a[2], ... a[n])

  여기서 $a[i] \le a[j]$ ($1 \le i \le j-1$), $a[j] \le a[k]$ ($j+1 \le k \le n$)
- 이때 a[j]을 분할 원소(또는 중추키)라 하며, 리스트에서 j 번째 작은 원소이다. 따라서 a[j]은 리스트가 정렬된 후에도 위치 변동이 없게된다.
- 퀵 정렬은 P1와 P3에 대해서 이러한 분할 과정을 반복한다.
- 리스트에 한 개의 원소만 존재할 때 Small(P) 가 true로 되며, 이 경우 이미 리스트는 정렬된 것으로 생각할 수 있으므로 바로 return 한다.

- 예:

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] p q
(26 5 37 1 61 11 59 15 48 19) +∞ 1 10
↑ ↑
i j
19 37
15 61
↑ ↑
j i
(11 5 19 1 15) 26 (59 61 48 37) +∞ 1 5
반복 적용
(1 5) 11 (19 15) 26 1 2
1 5
15 19 4 5
(59 61 48 37) +∞ 7 10
(48 37) 59 (61) +∞ 7 8
37 48
61 +∞ 10 10
1 5 11 15 19 26 37 48 59 61 ➔ 결과

분할에 의한 퀵 정렬(lec5-1)

```
void QuickSort(int p, int q)
  // Sorts the elements a[p],..., a[q] which reside in
  // the global array a[1:n] into ascending order; a[n+1]
  // is considered to be defined and must be >= all the
  // elements in a[1:n].
  {
      if (p < q) { // If there are more than one element
        // divide P into two subproblems.
          int j = Partition(a, p, q+1);
            // j is the position of the
            // partitioning element.
        // Solve the subproblems.
          QuickSort(p, j-1);
          QuickSort(j+1,q);
        // There is no need for combining solutions.
      }
  }
```

분할(lec5-1)

```
int Partition(int a[], int m, int p)
  // Within a[m], a[m+1],..., a[p-1] the elements
  // are rearranged in such a manner that if
  // initially t==a[m], then after completion
  // a[q]==t for some q between m and p-1, a[k]<=t
  // for m<=k<q, and a[k]>=t for q<k<p. q is returned.
  {
      int v=a[m]; int i=m, j=p;
      do {
            do i++;
            while (a[i] < v);
            do j--;
            while (a[j] > v);
            if (i < j) Interchange(a, i, j);
        } while (i < j);
        a[m] = a[j]; a[j] = v; return(j);
  }
```

비교 횟수 분석

- 분할의 각 호출에서 원소 비교 횟수는 많아야 $p-m+1$ 이다.

  r 을 순환 호출의 임의의 레벨에서 분할의 모든 호출에 포함된 원소들의 총 수라 하자. 레벨 1에서 $r = n$, 레벨 2에서 $r = n-1$, …. 그리고 각 레벨의 분할에서 $O(r)$ 원소 비교가 이루어진다.
- 최악의 경우 $C_W(n)$:

  각 레벨의 r 이 이전 레벨의 r 보다 1 만큼 감소할 때(예, 이미 정렬된 리스트)가 이 경우에 해당하며, 총 비교 횟수는

  $n + (n-1) + \cdots + 2$가 되고, $O(n^2)$ 이다.
- 평균의 경우 $C_A(n)$:

  분할 원소 v 가 a[m:p-1]에서 i ($1 \le i \le p-m$)번째 작은 원소가 될 확률이 모두 같으므로, 정렬하기 위해 남은 두 부분 배열이 a[m:j]와 a[j+1:p-1]이 될 확률은 $1/(p-m)$, $m \le j < p$ 이다. 따라서

  $C_A(n) = n+1 + 1/n \sum_{1 \le k \le n}[C_A(k-1) + C_A(n-k)]$ ➔ $O(n \log n)$

스택 공간

- 최악의 경우 순환 호출의 최대 깊이가 n-1 이 될 수 있으므로 O(n)이 될 수 있다.
- 분할 후 생성된 2개의 리스트 a[p:j-1]과 a[j+1:q] 중 더 짧은 리스트를 먼저 정렬함으로써 필요한 스택 공간을 O(log n)으로 줄일 수 있다.

  → 이 개념을 사용한 반복적 퀵 정렬: 프로그램 (lec5-2)

퀵 정렬의 반복적 버전(lec5-2)

```
import java.util.Stack;
void QuickSort2(int p, int q)
  // Sorts the elements in a[p:q].
  {
     Stack<Integer> stack = new Stack<Integer>();    // 자바 시스템  스택 클래스 사용
      do {
        while (p < q) {
          int j = Partition(a, p, q+1);
          if ((j-p) < (q-j)) {
            stack.push(j+1);
            stack.push(q); q = j-1;
          }
          else {
            stack.push(p);
            stack.push(j-1); p = j+1;
          }
        }; // Sort the smaller subfile.
        if (stack.isEmpty()) return;
        q = stack.pop();  p = stack.pop();
      } while (true);
  }
```

성능 측정

- 퀵 정렬과 합병 정렬의 성능 비교를 위해 모두 순환 알고리즘을 사용하였으며, 퀵 정렬에서 분할 원소는 중간값 규칙(median of three rule: 즉, 분할 원소는 a[m], a[(m+p-1)/2], a[p-1] 중의 중간값)을 사용하였다.
- 평균 계산 시간

n	1000	2000	3000	4000	5000
합병정렬	72.8	167.2	275.1	378.5	500.6
퀵 정렬	36.6	85.1	138.9	205.7	269.0

- 최악의 계산 시간

n	1000	2000	3000	4000	5000
합병정렬	105.7	206.4	335.2	422.1	589.9
퀵 정렬	41.6	97.1	158.6	244.9	397.8

2.5 선택(selection)

- 문제: 리스트 (a[1], a[2], ... a[n])과 k 가 주어질 때 k 번째 작은 원소를 결정하여(a[j]라 하자) k 위치에 두며, k 위치 앞에는 a[j] 보다 작거나 같은 원소들이, k 위치 뒤에는 a[j] 보다 크거나 같은 원소들이 오도록 재배치한다.

  예) 리스트 = (12, 30, 7, 49, 25)

  k = 1(첫번째 작은 원소): (7, 12, 30, 49, 25)

  k = 3(세번째 작은 원소): (12, 7, 25, 30, 49)
- 퀵 정렬에서 사용된 분할을 사용한다. 즉, 분할 원소 a[j]를 사용하여 분할하면

  P1 = (a[1], a[2], ... a[j-1])

  P2 = (a[j])

  P3 = (a[j+1], a[2], ... a[n]) 으로 된다.

  이때, j == k 이면 멈추고, 만약 k < j 이면 P1, k > j 이면 P3에 대해 계속한다.

선택(lec5-3)

```
void Select1(int a[], int n, int k)
  // Selects the kth-smallest element in a[1:n] and
  // places it in the kth position of a[]. The
  // remaining elements are rearranged such that
  // a[m] <= a[k] for 1 <= m < k, and
  // a[m] >= a[k] for k < m <= n.
  {
     int low=1, up=n+1;
     a[n+1] = Integer.MAX_VALUE ; // a[n+1] is set to infinity.
     do { // Each time the loop is entered,
           // 1<=low<=k<=up<=n+1.
       int j = Partition(a, low, up);
           // j is such that a[j] is the
           // jth-smallest value in a[].
       if (k == j) return;
       else if (k < j) up = j; // j is the new upper limit.
       else low = j+1; // j+1 is the new lower limit.
     } while (true);
  }
```

시간 분석

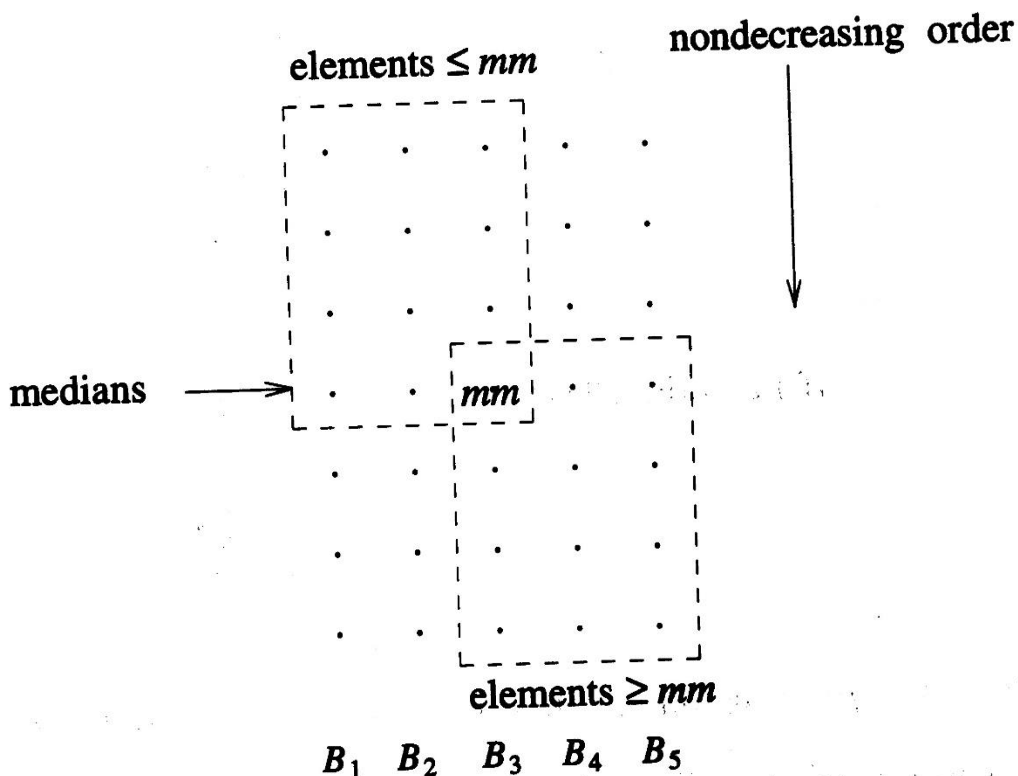
- Partition 함수는 O(p-m) 시간이 필요하다(즉, 원소의 수에 비례). 선택 알고리즘에서 최악의 경우 Partition이 n번 호출될 수 있다.

  예: 리스트의 원소들이 정렬된 상태에서 k = n 일 때(리스트의 첫 원소가 분할 원소로 사용됨)

  → $O(n^2)$
- 그러나 평균 계산 시간 $T_A(n)$은 O(n) 이다.
- 퀵 정렬과 마찬가지로 선택 알고리즘의 성능에 영향을 미치는 것은 분할 원소의 선택이다.

최악의 경우 최적 알고리즘(Select2)

- 분할 원소를 좀 더 신중하게 선택함으로써 최악의 경우 복잡도 O(n)을 갖는 선택 알고리즘을 만들 수 있다.
- 중간 값들의 중간 값 규칙(median of median(mm) rule)을 사용한다.



- 복잡한 분석을 통해 다음과 같은 식을 얻는다.
  $T(n) \le 72c_1n \quad \rightarrow O(n)$
- Select1 과 Select2의 성능 측정

n	1000	2000	3000	4000	5000
Select1	7.42	23.50	30.44	39.24	52.36
Select2	49.54	104.02	174.56	233.56	288.64

2.6 Strassen의 행렬 곱셈

- A와 B를 두 n x n 행렬이라 할 때, 곱 행렬 C = A·B 는 역시 n x n 행렬로서 다음과 같이 구할 수 있다.

  $C(i,j) = \Sigma_{1 \le k \le n} A(i,k) \cdot B(k,j)$
- 위 공식에서 C(i,j)의 계산은 n 번의 곱셈이 필요하고, 행렬 C는 n^2 개의 원소를 가지고 있으므로, 행렬 알고리즘에 대한 계산시간은 $O(n^3)$이다.
- 분할–정복 기법을 사용한 행렬 곱셈:

  $n = 2^k$ 라 하자.

  A와 B를 각각 네 개의 부분 정방 행렬로 분할하면, 각 부분 행렬은 n/2 x n/2 의 크기를 가지며 2 x 2 행렬 곱셈 공식을 이용하여 곱 A·B 를 계산할 수 있다.

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

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}$$

$$C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21}$$

$$C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

- n = 2 일 경우, 위 식은 원소들에 대한 곱셈 연산을 사용하여 계산할 수 있으며, n > 2 일 경우, C의 원소들은 n/2 x n/2 크기의 행렬에 대한 덧셈과 곱셈 연산을 사용하여 계산할 수 있다.
- A·B를 계산하기 위해 8번의 곱셈과 4번의 덧셈을 수행해야 하고, 덧셈은 cn^2 시간에 더해질 수 있다.

$$T(n) = \begin{cases} b & n \le 2 \\ 8T(n/2) + cn^2 & n > 2 \end{cases}$$

- 위 식을 풀면 $T(n) = O(n^3)$ 이며, 계산상의 이익이 없다.

Strassen의 곱셈 알고리즘

- 행렬 곱셈은 행렬 덧셈보다 시간이 더 많이 걸리므로, 곱셈 수를 한 번 줄이고 덧셈과 뺄셈을 좀 더 많이 사용하여 전체적인 시간 복잡도를 줄일 수 있다.
- 7번의 곱셈과 18번의 덧셈 및 뺄셈을 사용한다.

$$P = (A_{11} + B_{11})(B_{11} + B_{22})$$

$$Q = (A_{21} + A_{22})B_{11}$$

$$R = A_{11}(B_{12} - B_{22})$$

$$S = A_{22}(B_{21} - B_{11})$$

$$T = (A_{11} + A_{12})B_{22}$$

$$U = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$V = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$C_{11} = P + S - T + V$$

$$C_{12} = R + T$$

$$C_{21} = Q + S$$

$$C_{22} = P + R - Q + U$$

- Strassen 곱셈 알고리즘에 대한 순환 관계식:

$$T(n) = \begin{cases} b & n \le 2 \\ 7T(n/2) + an^2 & n > 2 \end{cases}$$

- 위 식을 풀면 T(n) = $O(n^{\log 7})$ = $O(n^{2.81})$ 이며, n 값이 매우 클 때 (즉, 큰 행렬의 곱셈) 시간을 절약할 수 있다.

레포트 #2

- n 개의 random number 들에 대해 합병정렬과 퀵정렬의 성능을 측정하여 표로 만들고, 그래프로 그려라.
- n = 1000, 5000, 10000, 20000, 50000, 100000에 대해 테스트하라.
- 각각의 n 에 대해 적어도 10 개의 테스트 데이터에 적용하고 그 평균을 산출하여 표와 그래프를 만들어라.
- 합병 정렬과 퀵 정렬은 순환호출 함수를 사용하라.