2 정보과학과 문제

이 교재에서는 지금까지 수학에서 다루어 온 문제들을 조금 다른 관점으로 분류하고 정의한다. 이 교재에서는 문제를 단순계산 문제, 결정 문제, 최적화 문제로 나누어 다룬다.

가. 계산 문제

정보과학에서 계산 문제(computational problem)는 수학적으로 계산 가능하며, 컴퓨터를 이용하여 풀 수 있는 모든 문제들을 의미한다.

예를 들어, "자연수 n이 주어질 때, n의 약수를 모두 구하시오." 와 같은 문제는 계산 문제이다. 이와 같이 계산 문제란 잘 정해진 규칙으로 인해 아무런 오해의 소지가 없이 정해진 방법으로 규칙을 적용하면 답을 구할 수 있는 모든 문제이다. 쉽게 말하자면 계산 가능한 문제를 의미한다. 여기서 말하는 계산이란 단순히 사칙연산의 수준을 벗어난다. 다음 예를 통하여 계산 문제와 그렇지 않은 문제를 이해하자.

[표-1] 계산이 가능한 문제와 계산이 불가능한 문제

계산이 가능한 문제	계산이 불가능한 문제
① N 이하의 자연 수 중 짝수의 개수는? ② 폐구간 $[a,\ b]$에서 소수는 존재하는가? ③ n개의 원소를 가지는 집합 S의 원소를 오름차순으로 나열하고, 그 최댓값을 구하시오. ④ N개의 원소로 이루어진 집합의 부분집합들 중 그 원소의 합이 k인 부분집합이 존재하는가? ⑤ 좌표평면 상에 N개의 점의 한 점에서 출발하여 모든 점을 지나고 출발점으로 돌아오는 경로의 길이 중 가장 짧은 길이는 얼마인가?	① 우리 반 학생들 중 키가 큰 학생은 몇 명인가? ② 주어진 음식들 중 맛있는 순서로 나열하시오. ③ 대한민국에서 축구를 잘하는 사람은 모두 몇 명인가? ④ 임의로 한 명을 골랐을 때, 그 사람이 요리를 잘 할 확률은 얼마인가?

일반적으로 정보과학에서는 계산 가능한 문제에 대해 결정 문제(decision problem), 탐색 문제(search problem), 카운팅 문제(couting problem), 최적화 문제(optimization problem),

함수형 문제(function problem) 등으로 나누지만, 이 책에서는 크게 "결정 문제"와 "최적화 문제"로 구분하여 다룬다.

일반적으로 정보올림피아드를 비롯한 각종 프로그래밍 대회에서는 최적화 문제나 NP－complete 문제[1]에서 입력값 n의 크기를 줄이거나, 제약조건을 두어 다루는 경우가 많다.

나. 결정 문제

결정 문제란 계산 문제들 중 그 결과를 'Yes', 'No' 중 하나로 답할 수 있는 문제를 의미한다. [표-1]에서 계산문제의 예시들 중 ②, ④와 같은 문제가 결정 문제이다.

결정 문제는 각종 프로그래밍 대회에서 직접적으로 출제되는 경우는 흔하지 않지만, 난이도가 높은 최적화 문제를 결정 문제의 형태로 바꾸어 해결하고, 그 결과들을 이용하여 해를 구하는 방법이 있다. 따라서 결정 문제를 간접적으로 다룰 줄 알아야 한다.

다. 최적화 문제

최적화 문제는 계산결과 얻은 후보 해들 중 가장 적절한 해를 찾는 형태의 문제를 말한다. [표-1]에서 계산 문제의 예시들 중 ③, ⑤와 같이 최댓값이나 가장 짧은 경로의 길이를 구하는 형태의 문제로 정보올림피아드를 비롯한 각종 프로그래밍 대회에서 가장 자주 출제되는 형태의 문제이다. 따라서 이 책에서는 최적화 문제를 많이 다룬다.

1) 결정 문제의 해를 검증하는 방법은 다항시간으로 가능하나, 해를 구하는 다항시간의 방법이 아직 알려지지 않은 결정 문제들의 집합으로, 3－SAT이라고 불리는 문제를 풀면 모두 풀 수 있음이 알려져 있다.

3 알고리즘과 실행시간 측정

가. 알고리즘

알고리즘의 정보과학적 정의에 대해서는 자세히 다루지 않겠지만 간단히 말하자면, 알고리즘이란 주어진 문제를 해결하기 위한 단계 혹은 절차를 말한다. 그리고 이 절차에는 입력값과 출력값이 존재해야하며, 유한한 단계를 거쳐서 반드시 종료되어야 한다.

일반적으로 프로그래밍 대회들에서는 이 유한한 단계를 제한시간으로 설정하여 주로 수초 이내에 작성한 알고리즘이 해를 구할 수 있는지 여부로 유한성을 판단한다.

알고리즘은 주로 자연어, 의사코드, 프로그래밍언어 등의 방법으로 기술할 수 있다. 다음은 집합 S의 원소들의 합을 구하는 알고리즘 A를 각 방법으로 기술한 예이다.

[자연어]

알고리즘 A
(1단계) 원소의 인덱스를 id로 정의한다.
(2단계) 집합 S에 대하여 $\sum_{id=1}^{n} S_{id}$를 구하고 이를 s라 한다.
(3단계) s를 출력하고 종료한다.

[의사코드]

알고리즘 A
(1단계) $id \leftarrow 1$, $s \leftarrow 0$
(2단계) $s = s + S_{id}$, $id \leftarrow id+1$
(3단계) $id \leq n$ $goto$ 2단계
(4단계) $print\ s$

[프로그래밍 언어 (C++)]

```
void A(int S[], int n)
{
  int s = 0;
  for(int id=1; id<=n; id++)
    s = s + S[id];
  printf("%d₩n",s);
}
```

이 책에서는 모든 알고리즘을 C++언어로 표현한다.

기본적으로 알고리즘에서 제시된 방법에 따라서 효율성이 달라진다. 알고리즘의 효율성을 측정하기 위하여 다양한 방법이 있지만 이 교재에서는 알고리즘의 효율성을 계산량으로 표현하며, 계산량은 입력크기 n에 대한 실행시간을 나타낸다.

위에서 제시된 알고리즘의 계산량은 입력크기 n이 커지면 실행시간은 n에 1차 함수적으로 비례하여 커진다는 것을 쉽게 알 수 있다. 따라서 위 알고리즘A의 계산량은 n이다. 그리고 이를 정보과학에서는 $O(n)$이라고 표현하기도 한다. 이와 같이 계산량을 나타내는 점근적 표현 방법으로 O, Θ, Ω 등이 있으나 이 교재에서는 따로 다루진 않는다.

나. 실행시간의 측정

정보올림피아드와 같은 각종 프로그래밍 경시대회나 세계적으로 운영되고 있는 Online Judge 등에서는 알고리즘의 성능을 주어진 입력에 대한 실행시간으로 측정한다.

일반적으로 실행시간은 CPU time만을 측정해야 하지만 정보올림피아드에서는 IO time까지 포함되기 때문에 입출력을 빠르게 작성하는 것도 어느 정도 도움이 된다.

실행시간을 측정하는 방법을 알아두면 알고리즘의 효율을 실험적으로 확인해 볼 수 있는데, 주어진 n개의 데이터를 정렬하는 문제를 이용해 실행 시간을 측정하는 방법에 대해서 알아본다.

먼저 무작위 정수 n개를 발생시키는 방법은 다음과 같다.

줄	코드	참고
1	#include <stdio.h>	5: 이 문제에서 입력값 n의 정의역은 3 <= n <= 100,000으로 한다. 9: 랜덤 시드 값을 시간으로 결정 12: rand()함수는 $0 \sim 2^k - 1$의 값을 발생(k의 값은 컴파일러에 따라 다르지만 보통 15, 31 중 하나임)
2	#include <stdlib.h>	
3	#include <time.h>	
4		
5	int n, S[100000];	
6		
7	int main()	
8	{	
9	srand(time(NULL));	
10	scanf("%d", &n);	
11	for(int i=0; i<n; i++)	
12	S[i] = rand();	
13	return 0;	
14	}	

위 방법으로 작성하면 배열 S에 랜덤한 값 n개가 입력된다. 다음 줄에 출력문을 추가해 값을 확인해 보자.

줄	코드	참고
1	#include <stdio.h>	
2	#include <stdlib.h>	
3	#include <time.h>	
4	int n, S[100000];	
5	int main()	
6	{	
7	srand(time(NULL));	
8	scanf("%d", &n);	
9	for(int i=0; i<n; i++)	
10	S[i] = rand();	
11	for(int i=0; i<n; i++)	
12	printf("%d ", S[i]);	
13	return 0;	
14	}	

[실행결과]

```
10
13426 30105 8303 21311 26182 6776 9981 29521 12570 26141
--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

다음으로 정렬 알고리즘을 작성하여 n개의 자료를 정렬하는 시간을 측정한다. 먼저 선택정렬(selection sort)을 작성하여 측정한다.

줄	코드	참고
1	`#include <stdio.h>`	
2	`#include <stdlib.h>`	
3	`#include <ctime>`	
4		
5	`int n, S[100000];`	
6		
7	`void print_array()`	
8	`{`	
9	`  for(int i=0; i<n; i++)`	
10	`    printf("%d ", S[i]);`	
11	`  printf("\n");`	
12	`}`	
13		
14	`void swap(int a, int b)`	
15	`{`	
16	`  int t=S[a];`	
17	`  S[a]=S[b];`	
18	`  S[b]=t;`	
19	`}`	
20		
21	`void selection_sort(void)`	
22	`{`	
23	`  for(int i=0; i<n-1; i++)`	
24	`    for(int j=i+1 ; j<n; j++)`	
25	`      if(S[i] > S[j])`	
26	`        swap(i, j);`	
27	`}`	
28		
29	`int main()`	

줄	코드	참고
30	{	
31	srand(time(NULL));	
32	scanf("%d", &n);	
33	for(int i=0; i<n; i++)	
34	S[i] = rand();	
35	//print_array();	
36	int start = clock();	
37		
38	selection_sort();	
39		
40	printf("result=%.3lf(sec)\n",(double)(clock()-start)/CLOCKS_PER_SEC);	
41	//print_array();	
42	return 0;	
43	}	

빨간색 부분이 알고리즘의 실행시간을 측정하는 부분이다. 즉, selection_sort 함수의 실행시간을 측정하는 프로그램이다. print_array 출력함수는 n값이 커지면 출력결과가 너무 많기 때문에 주석처리를 한 것이다. 실행결과는 다음과 같다.

[실행결과]

[n=1,000]

```
1000
result=0.002(sec)

--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

[n=10,000]

```
10000
result=0.337(sec)

--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

```
[n=100,000]
100000
result=28.078(sec)

--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

보는 바와 같이 선택정렬은 자료가 10배 증가할수록 실행시간은 약 100배 증가함을 알 수 있다. 따라서 n배 커지면 실행시간은 n^2에 비례하여 증가한다.

퀵정렬(quick sort)의 경우는 선택정렬보다 훨씬 빠른 속도로 정렬할 수 있는 알고리즘이다. 여기서는 퀵정렬 기반으로 동작하는 std::sort( )를 이용해 실행시간을 측정한다.

줄	코드	참고
1	`#include <stdio.h>`	4: std::sort를 사용하기 위하여 추가 6: 100만개 까지 측정
2	`#include <stdlib.h>`	
3	`#include <time.h>`	
4	`#include <algorithm>`	
5		
6	`int n, S[1000000];`	
7		
8	`void print_array()`	
9	`{`	
10	`  for(int i=0; i<n; i++)`	
11	`    printf("%d ", S[i]);`	
12	`  printf("\n");`	
13	`}`	
14		
15	`int main()`	
16	`{`	
17	`  srand(time(NULL));`	
18	`  scanf("%d", &n);`	
19	`  for(int i=0; i<n; i++)`	
20	`    S[i] = rand();`	
21	`//print_array();`	
22		
23	`  int start = clock();`	
24		
25	`  std::sort(S, S+n);`	

줄	코드	참고
26		
27	printf("result=%.3lf(sec)₩n", (double)(clock()-start)/CLOCKS_PER_SEC);	
28		
29	//print_array();	
30	}	

[n=100,000]

```
100000
result=0.016(sec)

--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

[n=1,000,000]

```
1000000
result=0.187(sec)

--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

이와 같이 100만개까지 정렬 하는데 0.2초 이내로 처리할 수 있으므로, 알고리즘 효율의 차이를 직접 체감할 수 있다.

C++에서는 위 예제에서 사용한 std::sort() 와 같이 프로그래밍 대회에서 사용할 수 있는 다양한 함수를 제공한다. 이러한 함수의 사용법은 교재에서 사용할 때마다 소개하니, 사용법을 익혀두면 다양한 알고리즘에 응용할 수 있으므로 꼭 익혀두기 바란다.

std::sort()는 $O(n \lg n)$으로 자료를 정렬하는 함수이며, 배열, 구조체, std::vector, std::list, std::set, std::map 등의 다양한 형태의 자료구조를 모두 정렬할 수 있는 매우 강력한 정렬 함수이다. 기본적인 활용방법은 다음과 같다.

```
std::sort(정렬할 자료의 시작 주소, 정렬할 자료의 마지막 주소, [비교함수의 주소]);
```

using namespace std; 명령을 실행한 다음이라면 std::를 생략하고 사용할 수 있다. 비교함수는 일반적으로 compare라는 이름으로 만들며, 생략하면 오름차순으로 정렬한다. 만약 내림차순이거나 구조체 등의 정렬에서 우선순위가 필요할 때는 비교함수를 작성해야 한다. 다음은 비교함수의 작성법을 보여준다.

[오름차순의 경우]

```
bool compare(int a, int b) //정수 배열의 오름차순 정렬일 경우
{
  return a < b;              //왼쪽의 원소가 오른쪽의 원소보다 값이 작도록 정렬
}
```

[내림차순의 경우]

```
bool compare(int a, int b) //정수 배열의 내림차순 정렬일 경우
{
  return a > b;              //왼쪽의 원소가 오른쪽의 원소보다 값이 크도록 정렬
}
```

[x, y를 멤버로 하는 POINT 구조체에서 x를 기준으로 오름차순 정렬]

```
bool compare(POINT a, POINT b) //POINT 구조체 정렬
{
    return a.x < b.x;          //x멤버 기준으로 오름차순
}
```

[x, y를 멤버로 하는 POINT 구조체에서 1순위 x, 2순위 y 기준 오름차순 정렬]

```
bool compare(POINT a, POINT b)     // POINT 구조체 정렬
{
    if( a.x == b.x ) return a.y < b.y;
    else return a.x < b.x;             // x멤버 기준으로 오름차순
}
```

위의 4가지 예시를 잘 이해하면 다양한 형태로 활용할 수 있다. 위와 같이 compare함수를 정의하면 std::sort()를 다음과 같이 사용하면 된다.

[S 배열의 처음부터 n-1번째까지의 원소를 compare 함수의 정의대로 정렬]

```
std::sort(S, S+n, compare);
```

알고리즘 학습을 위해서는 std::sort()를 활용하는 것도 중요하지만 퀵정렬과 같은 알고리즘을 직접 구현해보는 연습도 반드시 필요하다.

실제 대회 중에는 주어진 시간 동안 정확한 알고리즘을 만들어내야 하므로, 일반적으로 STL(Standard Template Library)를 활용할 줄 아는 것도 중요하다. STL이란 앞에서 언급한 sort, list, vector, set, map과 같은 표준 템플릿 라이브러리를 말한다.

마지막으로 실제로 퀵정렬을 직접 작성해 실행시간을 측정해 보자. 퀵정렬 알고리즘의 큰 틀은 다음과 같다.

1. pivot(기준값)을 정한다.
2. pivot보다 작은 원소들은 왼쪽으로, pivot보다 큰 원소들은 오른쪽으로 보낸다.
3. pivot을 기준으로 왼쪽 배열과 오른쪽 배열을 새로운 배열로 정하고, 각 배열 구간에 대해서 1번 과정을 재귀적으로 반복한다.

퀵정렬을 구현하는 여러 가지 방법들이 있다. 특히 1번 과정에서처럼 pivot(기준값)을 정하는 여러 가지 아이디어들이 있으나, 일반적으로 처음의 원소 또는 가장 마지막 원소를 pivot으로 잡는 방법을 사용한다.

하지만 이렇게 pivot을 잡았을 때 최악의 경우, 즉 역순으로 정렬되어 있을 경우 계산량이 $O(n^2)$이 될 수 있다. 하지만 이런 경우는 $\frac{1}{n!}$의 확률로 거의 발생하지 않는다. 만약 발생한다 하더라도 처음에 한 번 검사하여 그냥 뒤집으면 되기 때문에 일반적으로 큰 문제가 되지는 않는다.

수학적으로 이러한 경우가 절대로 발생하지 않도록 pivot을 잡는 다양한 방법이 있지만 이 예제에서는 단순히 처음에 나오는 원소를 pivot으로 잡는 방법으로 구현한다. 소스코드는 다음과 같다.

줄	코드	참고
1	void swap(int a, int b)	
2	{	
3	int t = S[a];	
4	S[a] = S[b];	
5	S[b] = t;	
6	}	
7		
8	void quick_sort(int s, int e)	
9	{	
10	if(s<e)	
11	{	
12	int p = s, l = s+1, r = e;	
13	while(l<=r)	
14	{	
15	while(l<= e && S[l]<=S[p]) l++;	
16	while(r>=s+1 && S[r]>=S[p]) r--;	
17	if(l<r) swap(l,r);	
18	}	
19	swap(p, r);	
20	quick_sort(s, r-1);	
21	quick_sort(r+1, e);	
22	}	
23	}	

직접 작성한 퀵정렬의 시간을 측정해보자.

줄	코드	참고
1	#include <stdio.h>	4: std::sort를 사용하기 위하여 추가 6: 100만 개 까지 측정
2	#include <stdlib.h>	
3	#include <time.h>	
4	#include <algorithm>	
5		
6	int n, S[1000000];	
7		
8	void print_array()	
9	{	
10	for(int i=0; i<n; i++)	
11	printf("%d ", S[i]);	
12	printf("\n");	
13	}	

줄	코드	참고
14		
15	void swap(int a, int b)	
16	{	
17	int t = S[a];	
18	S[a] = S[b];	
19	S[b] = t;	
20	}	
21		
22	void quick_sort(int s, int e)	
23	{	
24	if(s<e)	
25	{	
26	int p = s, l = s+1, r = e;	
27	while(l<=r)	
28	{	
29	while(l<= e && S[l]<=S[p]) l++;	
30	while(r>=s+1 && S[r]>=S[p]) r--;	
31	if(l<r) swap(l,r);	
32	}	
33	swap(p, r);	
34	quick_sort(s, r-1);	
35	quick_sort(r+1, e);	
36	}	
37	}	
38		
39	int main()	
40	{	
41	srand(time(NULL));	
42	scanf("%d", &n);	
43	for(int i=0; i<n; i++)	
44	S[i] = rand();	
45	//print_array();	
46		
47	int start = clock();	
48		
49	quick_sort(0, n-1);	
50		
51	printf("result=%.3lf(sec)\n",(double)(clock()-start)/CLOCKS_PER_SEC);	
52		
53	//print_array();	
54	return 0;	
55	}	

[n=100,000]

```
100000
result=0.016(sec)

--------------------------------
Process exited with return value 0
Press any key to continue . . .
```

[n=1,000,000]

```
1000000
result=0.206(sec)

--------------------------------
Process exited with return value 0
Press any key to continue . . .
```