11_2 More Recursion

Recursion 두번째 강의 시작하겠습니다.

The Recursive Method power

```
public class XToThePowerN {
    public static void main(String[] args) {
        for (int n = 0; n < 4; n++) {
            System.out.println("3 to the power " + n + " is " + power(3,n));
        }
    }
    public static int power(int x, int n) {
        if (n < 0) {
            System.out.println("Illegal argument to power");
            System.exit(0);
        }
        if (n > 0) // x^n = x^(n-1) * x (recursive call)
            return power(x, n-1) * x;
        else // n == 0, (stopping case)
            return 1;
    }
}
```

```
3 to the power 0 is 1
3 to the power 1 is 3
3 to the power 2 is 9
3 to the power 3 is 27
```

2

페이지 2

이번에는 x의 n승의 계산을 recursive method로 해 보겠습니다.
main method에서는 3의 0승부터 3의 3승 까지 차례로 구하도록
recursive method power(3, n) 을
n을 0부터 3까지 증가시키면서 call하였습니다.
제대로 계산이 된다면 이와 같이 output이 나와야 합니다.
이제 recursive method인 power(int x, int n) 을 보겠습니다.
먼저 n이 0보다 작은지를 test합니다.
n이 음의 정수라도 1/3, 1/9, ... 과 같이 계산이 됩니다만
여기서는 n이 음의 정수인 경우는 고려하지 않기로 하고
프로그램을 끝내게 됩니다.
n이 1 이상인 경우에는 recursive call case를 실행합니다.
x의 n승은 x의 (n 마이너스 1) 승 곱하기 n이므로
power(x, n-1) 을 call하여 나온 결과에 x를 곱하여 return합니다.
stopping case는 n이 0인 경우로
1을 return하면 됩니다.
이와 같이 recursive method에는
stopping case와 recursive call이 반드시 존재해야 합니다.

Evaluating the Recursive Method Call `power(2,3)`

1) power(2,3) = power(2,2) * 2
2) power(2,2) = power(2,1) * 2
3) power(2,1) = power(2,0) * 2
4) power(2,0) = 1,
5) power(2,1) = power(2,0) * 2 = 1 * 2 = 2, 4)를 3)의 power(2,0)에 대입
6) power(2,2) = power(2,1) * 2 = 2 * 2 = 4, 5)를 2)의 power(2,1)에 대입
7) power(2,3) = power(2,2) * 2 = 4 * 2 = 8, 6)을 1)의 power(2,2)에 대입

3

페이지 3

이제 power(2, 3)의 recursive call이 어떻게 실행되는지 보기로 하겠습니다.
먼저 1)에서 power(2, 3) 안에서 power(2,2)를 call합니다.
2)에서 power(2,2) 안에서 power(2,1)을 call하고
3)에서 다시 power(2,1) 안에서 power(2,0)을 call합니다.
4)에서 power(2,0)은 stopping case로 1을 return 합니다.
5)에서 4)의 결과를 3)의 power(2,0)에 대입하면
power(2,1)은 1 곱하기 2는 2가 됩니다.
6)에서 5)의 결과를 2)의 power(2,1)에 대입하면
power(2,2)는 2 곱하기 2는 4가 됩니다.
7)에서 6)의 결과를 1)의 power(2,2)에 대입하면
power(2,3)은 4곱하기 2는 8이 됩니다.

Recursive Design Techniques (Checking Steps)

1. Confirm there is no infinite recursion
2. Confirm each stopping case performs the correct action for that case
3. Confirm if all recursive calls perform their actions correctly, then the entire case performs correctly

페이지 4

문제를 풀 때 recursive 방법을 사용하여 solution을 디자인 한 경우
이 solution이 맞는 것인지를 체크하는 방법에 대해 알아보겠습니다.
먼저 recursion이 무한히 반복되지 않는다는 것을 확인해야 합니다.
두번째로 stopping case 각각이 올바른 action을 실행하는 것을 확인해야 합니다.
세번째로는 모든 recursive call들이 올바른 action을 실행하고,
그것들을 합친 전체 case가 올바른 동작을 함을 확인해야 합니다.

Binary Search

- Searching an array to find a given value
- Condition: the array should be a sorted array:
  a[0] ≤ a[1] ≤ a[2] ≤ . . . ≤ a[finalIndex]
- If the value is found, its index is returned
- If the value is not found, -1 is returned
- Implemented using recursion
  - Recursive method reduces the search space by about a half
  - "Divide and Conquer" technique

5

페이지 5

Binary search를 recursive 방식으로 디자인해 보고
이 solution이 옳은 것인지를 앞에서 언급한 방법으로
검증해 보도록 하겠습니다.
Binary search 문제는 array에서 주어진 value가 있는지
찾는 문제입니다.
Binary search 알고리즘을 사용하기 위해서는
array의 data가 sorting 되어 있어야 합니다.
편의상 오름차순으로 sorting되어 있다고 가정합니다.
search를 통해 value가 발견되면
그 value의 index를 return합니다.
발견이 안되는 경우에는 -1을 return합니다.
Binary search를 recursion을 이용하여 구현할 때
recursive call마다 search space를
절반 정도로 줄여서 search하게 됩니다.
이러한 방식을 "divide and conquer" 라고 부릅니다.

Execution of search method

search(first=0, last=9, key=63) mid=(0+9)/2=4 a[4]=57 < 63	search(first=mid+1=5, last=9, key=63) mid=(5+9)/2=7 a[7]=80 > 63	search(first=5, last=mid-1=6, key=63) mid=(5+6)/2=5 a[5]==63
a[0] = 15 a[1] = 20 a[2] = 35 a[3] = 41 a[4] = 57 a[5] = 63 a[6] = 75 a[7] = 80 a[8] = 85 a[9] = 90	a[0] = 15 a[1] = 20 a[2] = 35 a[3] = 41 a[4] = 57 a[5] = 63 a[6] = 75 a[7] = 80 a[8] = 85 a[9] = 90	a[0] = 15 a[1] = 20 a[2] = 35 a[3] = 41 a[4] = 57 a[5] = 63 a[6] = 75 a[7] = 80 a[8] = 85 a[9] = 90

6

페이지 6

Binary search를 실행하는 search method의 실행을
그림으로 정리하였습니다.
search method에는 first, last, key의
세개의 parameter가 주어집니다.
first와 last는 각각 array의 search 범위를 나타내는
시작 index와 끝 index 입니다.
key는 search에서 찾는 value입니다.
처음 search를 call할 때는 array의 전 범위를 커버하도록
first는 0, last는 a.length – 1로 주면 됩니다.
우리의 example에서는 first는 0, last는 9로 했고
target value인 key는 63이 주어졌습니다.
search method에서는 first와 last를 더해 반으로 나눈 값을
mid index로 계산하고
array a의 a[mid] 값이 key와 같은지를 테스트합니다.
우리의 example에서는 mid가 4이고
a[4]는 57로 key 63보다 작습니다.
이렇게 a[mid]가 key보다 작은 경우

우리는 search의 범위를 mid + 1부터 last까지로 좁힐 수 있습니다.
따라서 recursive call은 search(5, 9, 63) 이 되겠습니다.
이 경우 mid는 7이 되고 a[7]은 80으로 key보다 큽니다.
이 경우에는 search의 범위를 first부터 mid-1 까지로 좁힐 수 있으므로
recursive call은 search(5, 6, 63) 이 되겠습니다.
mid는 5가 되고 a[5]의 value가 63이므로
원하던 key를 찾은 경우입니다.
이제 key 값을 가진 index인 5를 return하면 됩니다.

No Existence Case

search(0, 9, 37) mid=(0+9)/2=4 a[4]=57 > 37	search(0, 3, 37) mid=(0+3)/2=1 a[1]=20 < 37	search(2, 3, 37) mid=(2+3)/2=2 a[2]=35 < 37	search(3, 3, 37) mid=(3+3)/2=3 a[3]=41 > 37	search(3, 2, 37) first(3) > last(2) not exist
a[0] = 15	a[0] = 15	a[0] = 15	a[0] = 15	a[0] = 15
a[1] = 20	a[1] = 20	a[1] = 20	a[1] = 20	a[1] = 20
a[2] = 35	a[2] = 35	a[2] = 35	a[2] = 35	a[2] = 35
a[3] = 41	a[3] = 41	a[3] = 41	a[3] = 41	a[3] = 41
a[4] = 57	a[4] = 57	a[4] = 57	a[4] = 57	a[4] = 57
a[5] = 63	a[5] = 63	a[5] = 63	a[5] = 63	a[5] = 63
a[6] = 75	a[6] = 75	a[6] = 75	a[6] = 75	a[6] = 75
a[7] = 80	a[7] = 80	a[7] = 80	a[7] = 80	a[7] = 80
a[8] = 85	a[8] = 85	a[8] = 85	a[8] = 85	a[8] = 85
a[9] = 90	a[9] = 90	a[9] = 90	a[9] = 90	a[9] = 90

7

페이지 7

이번에는 주어진 key 값이 array에 존재하지 않는 경우를 보겠습니다.
search(0, 9, 37)이 첫 call이고
mid는 4이고 a[4]는 57로 37보다 크기 때문에
search(0, 3, 37)을 recursive call합니다.
mid는 1이 되고, a[1]은 20으로 37보다 작으므로
search(2, 3, 37)을 recursive call하게 됩니다.
mid는 2가 되고, a[2]는 35로 37보다 작으므로
search(3, 3, 37)을 recursive call하게 됩니다.
mid는 3이 되고, a[3]은 41로 37보다 크므로
search(3, 2, 37)을 recursive call하게 됩니다.
여기에서 first인 3이 last인 2보다 큽니다.
즉, 주어진 search range가 valid하지 않게 되고
search하던 key는 이 array에 존재하지 않는다는
조건이 충족되게 됩니다.
따라서 -1을 return하게 됩니다.

Recursive Method for Binary Search

```
public static int search(int[] a, int first, int last, int key) {
    int result = 0;
    if (first > last) result = -1;  // stopping case
    else {
        int mid = (first + last)/2;
        if (key == a[mid]) result = mid; // stopping case
        else if (key < a[mid])
            result = search(a, first, mid-1, key); // recursive call
        else if (key > a[mid])
            result = search(a, mid+1, last, key);  // recursive call
    }
    return result;
}
```

8

페이지 8

지금까지 앞에서 보았던 과정을 프로그램 코드로 다시 썼습니다.
search method에는 array a의 reference와 first, last index
그리고 target인 key값이 parameter로 주어집니다.
result 변수는 최종적으로 답이 될 index (또는 -1)을 가지게 됩니다.
이 method에는 stopping case가 두 개 있는데
첫번째는 key value가 array에 존재하지 않는 경우입니다.
first가 last보다 커지면 이 경우를 만족하게 되고
result에 -1을 assign합니다.
두번째 stopping case는 key값이 array에 존재하는 경우입니다.
a[mid]의 value가 key와 같은 경우인데
이 때에는 result에 mid index를 assign합니다.
recursive call도 두 개가 있는데
그 첫번째는 key 값이 a[mid] 보다 작은 경우입니다.
이 때에는 search(a, first, mid-1, key) 를 recursive call하여
그 return 값을 result에 assign합니다.
두번째 recursive call은 key값이 a[mid] 보다 큰 경우로
이 때에는 search(a, mid+1, last, key)를 recursive call하여

그 return 값을 result에 assign합니다.
마지막으로 위에서 구해진 result를 return 하면 됩니다.

Checking the search Method (1/3)

1. There is no infinite recursion
   - On each recursive call
     - The value of **first** is increased
     - The value of **last** is decreased
   - So, eventually the method will be called with **first** larger than **last**

```
public static int search(int[] a, int first, int last, int key) {
    int result = 0;
    if (first > last) result = -1;  // stopping case
    else {
        int mid = (first + last)/2;
        if (key == a[mid]) result = mid; // stopping case
        else if (key < a[mid])
            result = search(a, first, mid-1, key); // recursive call
        else if (key > a[mid])
            result = search(a, mid+1, last, key);  // recursive call

    }
    return result;
}
```

9

페이지 9

이제 우리가 작성한 search method가 올바른 solution 인지를 체크해 보겠습니다.
이 체크는 4 페이지에서 언급한 세가지 스텝으로 진행합니다.
첫번째로 이 recursion이 무한 반복되지 않는다는 것을 보입니다.
만약 key가 array에 존재한다면
mid를 return하는 stopping case에 의해
method는 무한 반복되지 않습니다.
key가 array에 존재하지 않는 경우
각각의 recursive call마다
first의 value는 그 값을 유지하거나 mid + 1로 증가하게 됩니다.
또한 last의 value는 그 값을 유지하거나 mid – 1로 감소하게 됩니다.
따라서 언젠가는 first가 last보다 커지는 때가 오게 되어 있습니다.
즉 first가 last보다 커지는 stopping case에 의해
-1을 return하게 되고
따라서 무한 반복은 일어나지 않습니다.

Checking the `search` Method (2/3)

2. Each stopping case performs the correct action for that case
   - If **`first > last,`** there are no array elements between **`a[first]`** and **`a[last]`**, so **`key`** is not in this segment of the array, and **`result`** is correctly set to **`-1`**
   - If **`key == a[mid]`**, **`result`** is correctly set to **`mid`**

```
public static int search(int[] a, int first, int last, int key) {
    int result = 0;
    if (first > last) result = -1;  // stopping case
    else {
        int mid = (first + last)/2;
        if (key == a[mid]) result = mid; // stopping case
        else if (key < a[mid])
            result = search(a, first, mid-1, key);
        else if (key > a[mid])
            result = search(a, mid+1, last, key);
    }
    return result;
}
```

10

페이지 10

두번째 스텝으로 stopping case들이 올바르게 동작한다는 것을 보입니다.
search method에는 두 개의 stopping case가 있습니다.
먼저 first 가 last보다 클 때에는 array에 search 구간을 잡을 수가 없습니다.
따라서 key가 이 array에 존재할 수 없으므로
-1을 return합니다.
두번째로 a[mid]가 key와 같을 때에는
key값을 찾은 것이므로 mid를 return합니다.
따라서 두 경우 모두 올바르게 동작하고 있습니다.

Checking the `search` Method (3/3)

3. Check all recursive calls perform their actions correctly
   - If key < a[mid], then key must be one of the elements a[first] through a[mid-1], or it is not in the array, so we should search a from first to mid – 1.
   - If key > a[mid], then key must be one of the elements a[mid-1] through a[last], or it is not in the array, so we should search a from mid + 1 to last.

```
public static int search(int[] a, int first, int last, int key) {
    int result = 0;
    if (first > last) result = -1;  // stopping case
    else {
        int mid = (first + last)/2;
        if (key == a[mid]) result = mid; // stopping case
        else if (key < a[mid])
            result = search(a, first, mid-1, key);
        else if (key > a[mid])
            result = search(a, mid+1, last, key);
    }
    return result;
}
```

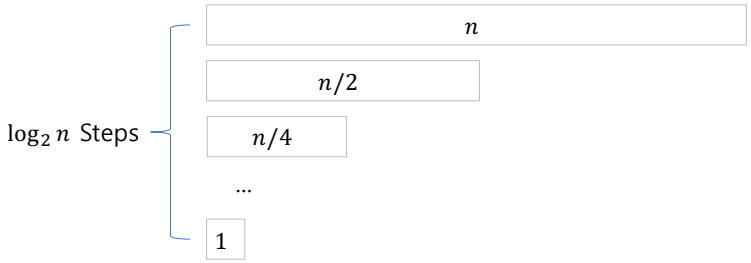
11

페이지 11

세번째로 모든 recursive call들이 올바르게 동작한다는 것을 보입니다.
이 method에는 두 개의 recursive call이 있습니다.
첫번째 case에 key값이 a[mid]보다 작을 때에는
array a가 sorting 되어 있기 때문에 a[mid]부터 a[last]까지의 값은
모두 key보다 큰 값이 됩니다.
따라서 search의 범위를 first부터 mid-1로 줄이는 것이 타당합니다.
만일 key가 array에 존재한다 해도 이 줄어든 범위에 있을 것입니다.
두번째 case에서 key값이 a[mid]보다 클 때에는
array a가 sorting 되어 있기 때문에 a[first]부터 a[mid]까지의 값은
모두 key보다 작은 값이 됩니다.
따라서 search의 범위를 mid+1부터 last로 줄이는 것이 타당합니다.
만일 key가 array에 존재한다 해도 이 줄어든 범위에 있을 것입니다.
지금까지 세가지 조건이 모두 만족함을 보였고,
따라서 Binaary search를 위한 recursive method는
정상적으로 동작한다는 것을 보이게 되었습니다.

Efficiency of Binary Search

- Array size = n
- Serial search algorithm
  - time complexity: $O(n)$ ... we should see all n elements in the worst case
- Binary search algorithm
  - time complexity: $O(\log n)$ ... we should see $\log(n)$ elements in the worst case



12

페이지 12

마지막으로 Binary Search의 efficiency에 대해 알아보겠습니다.
array size를 n이라 할 때
참고로 array를 serial search한다고 하면
최악의 경우 첫 element인 a[0]부터 끝 element인 a[a.length – 1] 까지
모두 보아야 하기 때문에
worst case의 time complexity는 order of n, 즉, O(n)이 됩니다.
Binary search의 경우는
worst case의 time complexity가 O(log n) 이 되어
이것은 serial search의 O(n) 보다 훨씬 빠릅니다.
그림에서는 n개의 search 범위의 size가
절반씩 줄어들면서 n/2, n/4, ..., 1 과 같이 되고
최악의 경우 search 범위의 size가 1까지 간다해도
그것은 log_2 n step만 걸리게 된다는 것을
보여주고 있습니다.