

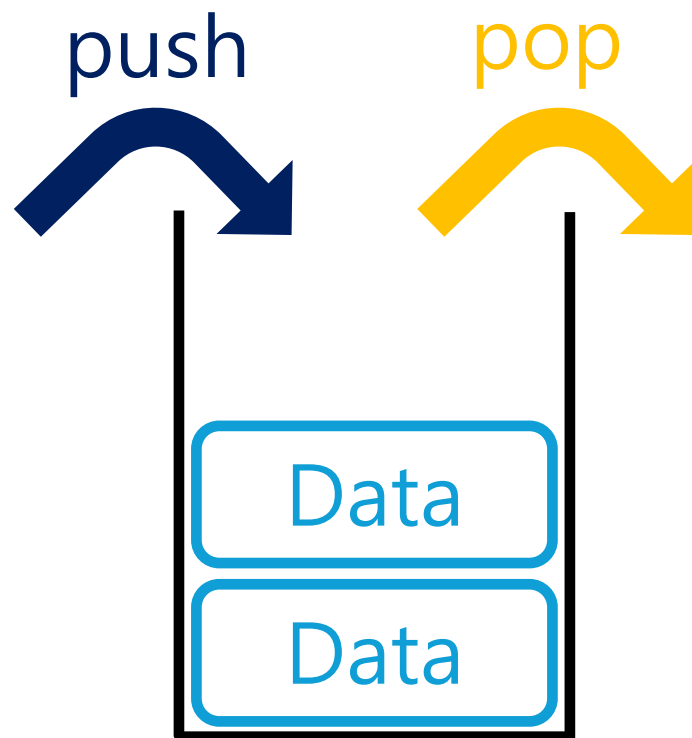
# Stack

즐겁고 알찬 자료구조 튜터링

# Stack

# Stack

- 먼저 들어간 것이 나중에 나오는 LIFO(Last-in, First-out)구조의 자료 구조
- 스택의 기본적인 연산
  - push
  - pop

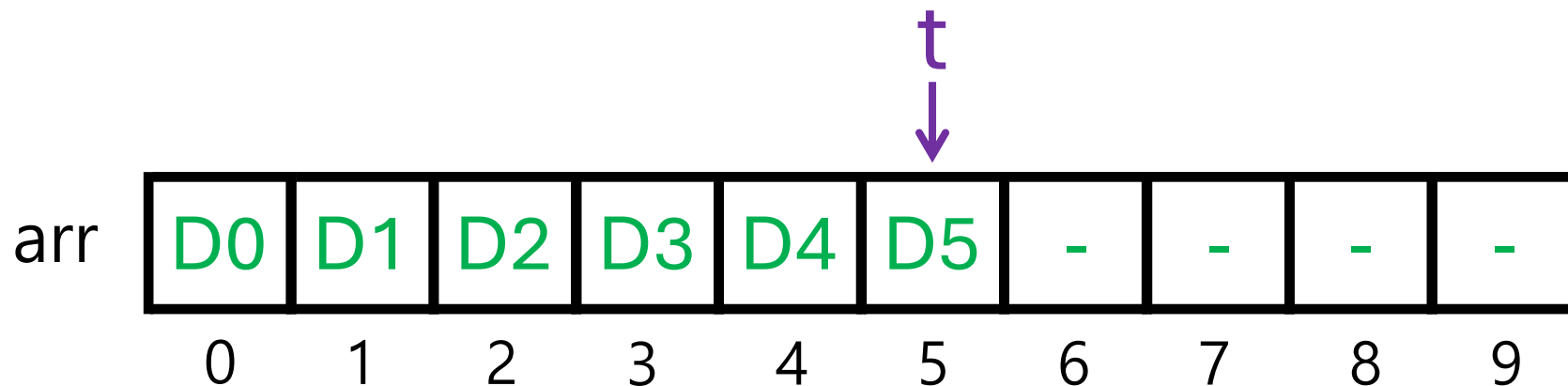


# Stack의 ADT

- `int size()` : 현재 stack에 있는 원소 개수를 반환
- `bool empty()` : stack에 원소가 비어있는지 여부를 반환
- `T& top()` : stack에서 마지막으로 저장된 원소를 반환
- `void push(T &data)` : stack에 인자로 받은 `data`를 저장하고, stack의 용량을 초과한 경우 예외 발생
- `void pop()` : stack에 저장된 마지막 요소를 삭제하고, stack이 비어있는 경우 예외 발생

# 배열 기반의 Stack

- 클래스의 멤버 변수로 **배열**을 가지고, 이를 Stack으로 사용
- 멤버 변수 **t(top)**을 활용하여 **마지막에 추가된 원소**를 가리킴



# 배열 기반의 Stack



```
1  template <typename T>
2  class ArrayStack {
3  private:
4      T *arr;
5      int cap;
6      int t;
```

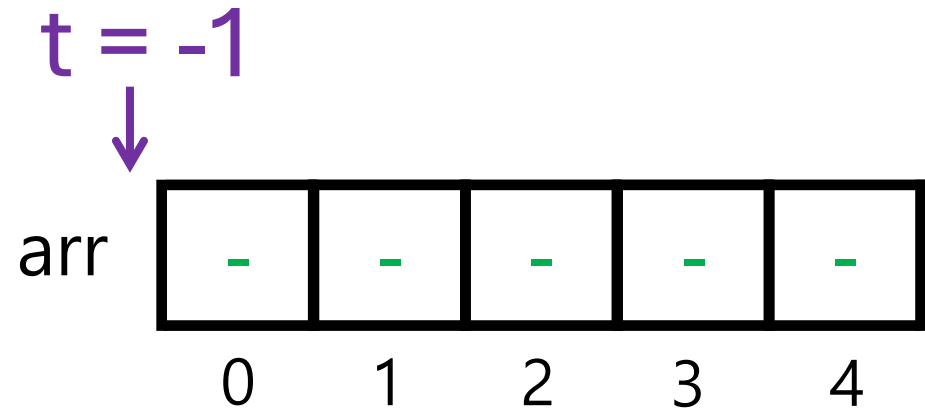


```
1  ArrayStack(int cap) {
2      this->cap = cap;
3      arr = new T[cap];
4      t = -1;
5  }
```

# size, empty



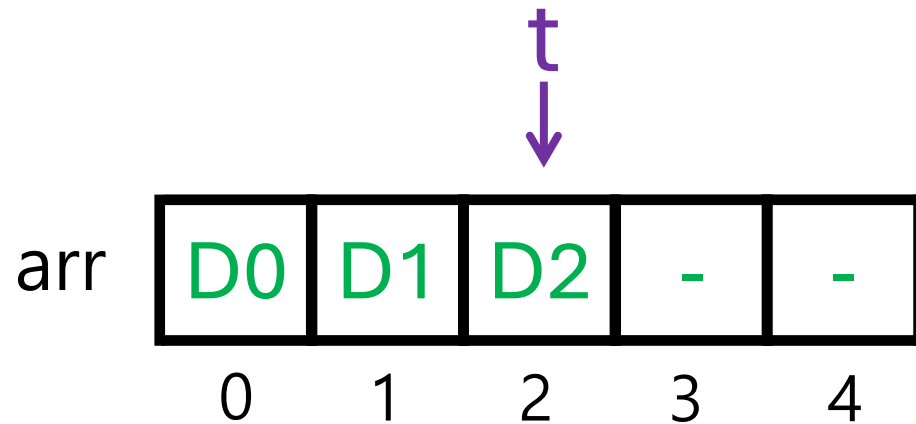
```
1 int size() const {  
2     return t + 1;  
3 }  
4  
5 bool empty() const {  
6     return t == -1;  
7 }
```



# top, push, pop

```
● ● ●  
1 T &top() const {  
2     if (empty()) {  
3         throw "Stack is empty";  
4     }  
5  
6     return arr[t];  
7 }
```

```
● ● ●  
1 void push(const T &data) {  
2     if (size() == cap) {  
3         throw "Stack is full";  
4     }  
5  
6     arr[++t] = data;  
7 }
```



```
● ● ●  
1 void pop() {  
2     if (empty()) {  
3         throw "Stack is empty";  
4     }  
5  
6     --t;  
7 }
```



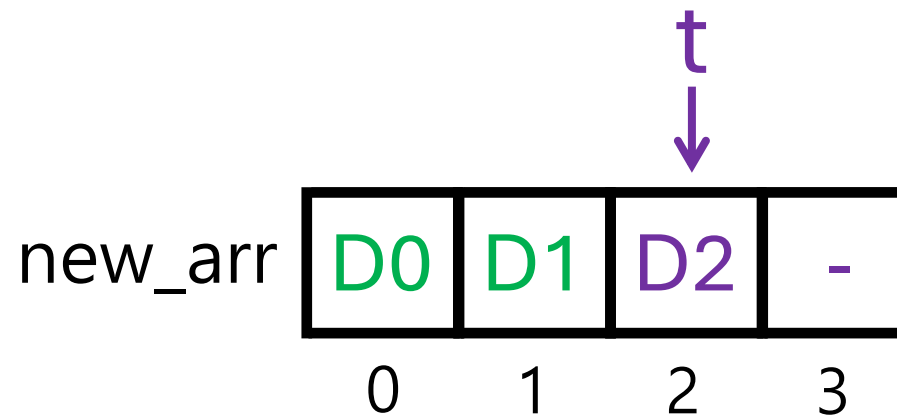
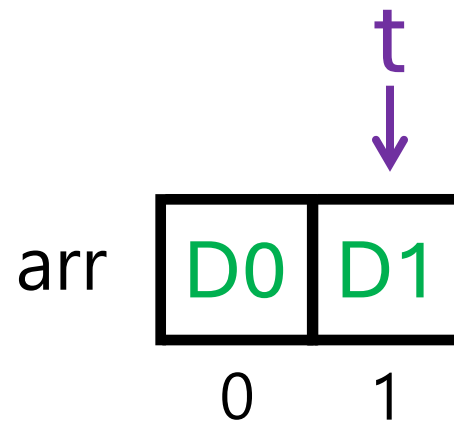
# 배열의 길이가 증가하는 Stack

- 배열 기반으로 stack을 구현한 경우 **배열 크기만큼** 원소를 push하지 못함
- push 연산에서 배열의 크기 이상으로 원소를 push하는 경우 **배열의 크기를 2배**로 늘려 새로운 배열 할당
- 기존의 배열에 있는 원소를 크기가 증가한 **새로운 배열**에 **옮겨주고**, 새로운 배열을 stack으로 사용

# 예외 없는 push



```
1 void push(const T &data) {  
2     if (size() == cap) {  
3         cap *= 2;  
4         T *new_arr = new T[cap];  
5         for (int i = 0; i <= t; i++) {  
6             new_arr[i] = arr[i];  
7         }  
8  
9         delete[] arr;  
10        arr = new_arr;  
11    }  
12  
13    arr[++t] = data;  
14 }
```



# Usage of Stack

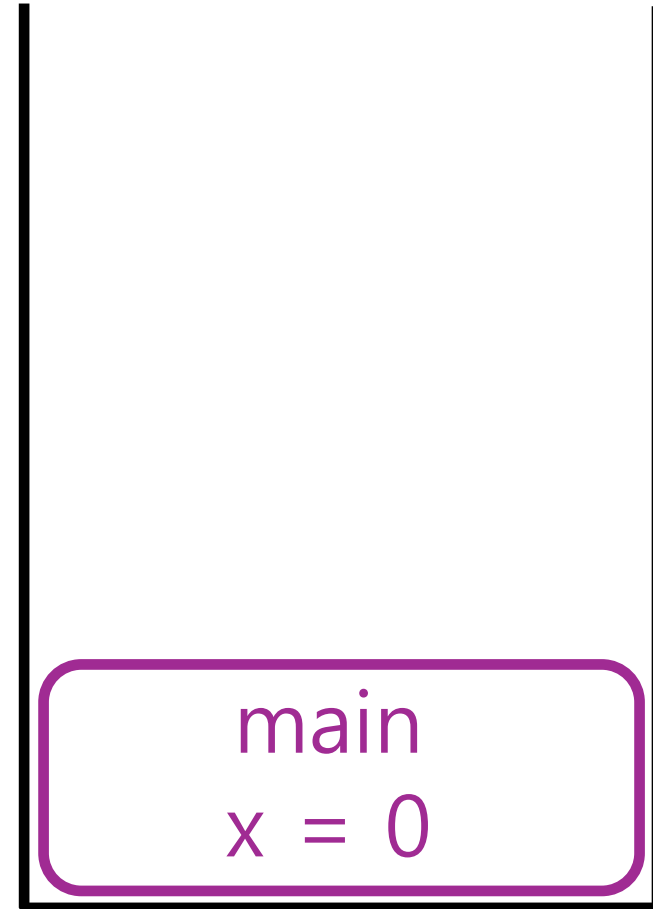
# Run-Time Stack

- 함수가 실행되면, 함수들은 메모리 공간에 할당
- 함수가 호출되는 순서에 따라 Run-Time Stack에 함수에 필요한 정보(지역변수, 반환값, Program Counter)들을 push
- 함수가 종료되면 Run-Time Stack에서 함수를 pop
- 따라서 현재 실행중인 함수는 Run-Time Stack의 top에 위치

# Run-Time Stack



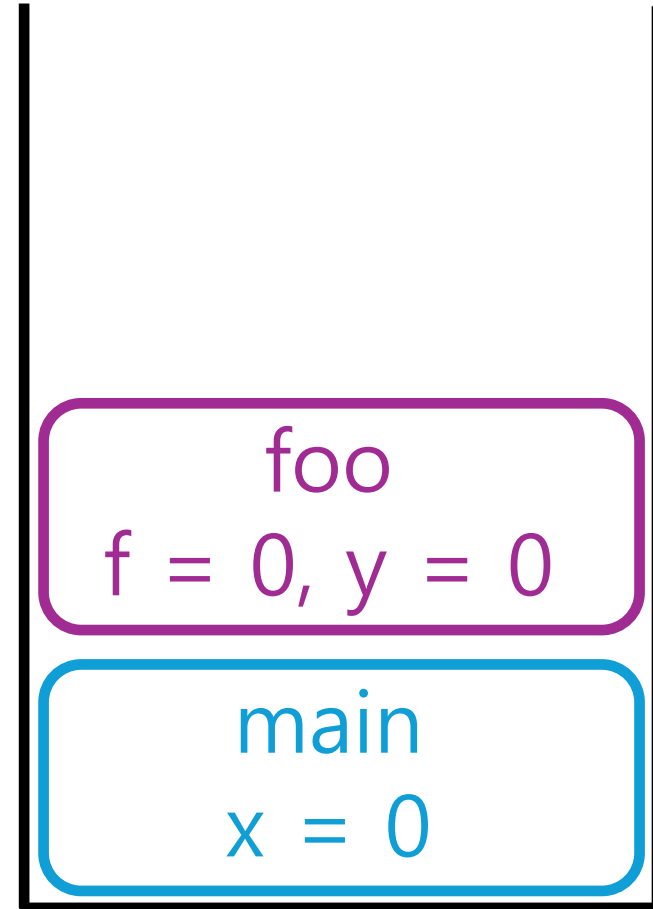
```
1  int main() {  
2      int x = 0;  
3      foo(x);  
4  }  
5  void foo(int f) {  
6      int y = f;  
7      bar(f);  
8  }  
9  void bar(int b) {  
10     int z = b;  
11 }
```



# Run-Time Stack



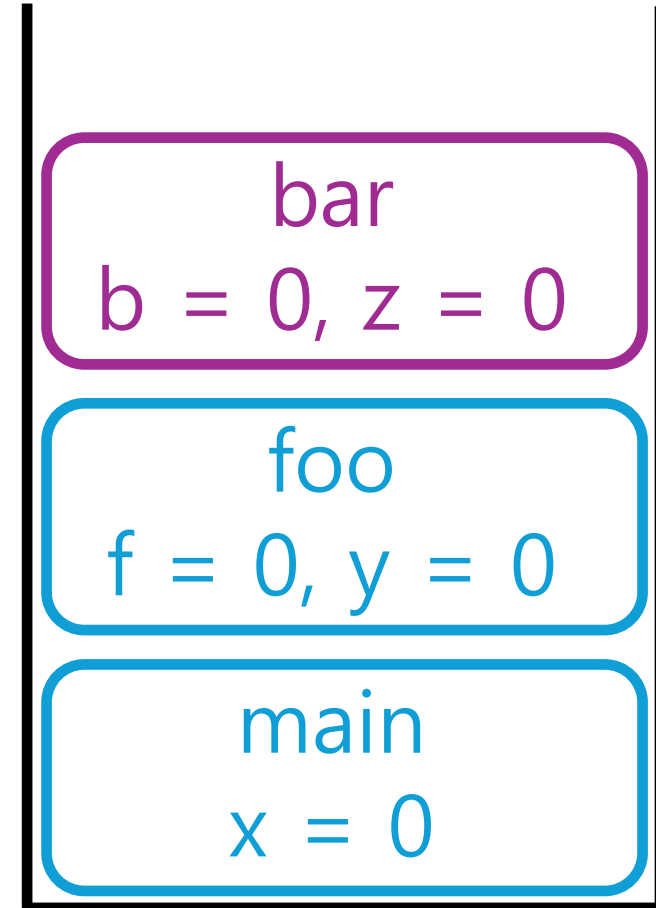
```
1  int main() {  
2      int x = 0;  
3      foo(x);  
4  }  
5  void foo(int f) {  
6      int y = f;  
7      bar(f);  
8  }  
9  void bar(int b) {  
10     int z = b;  
11 }
```



# Run-Time Stack



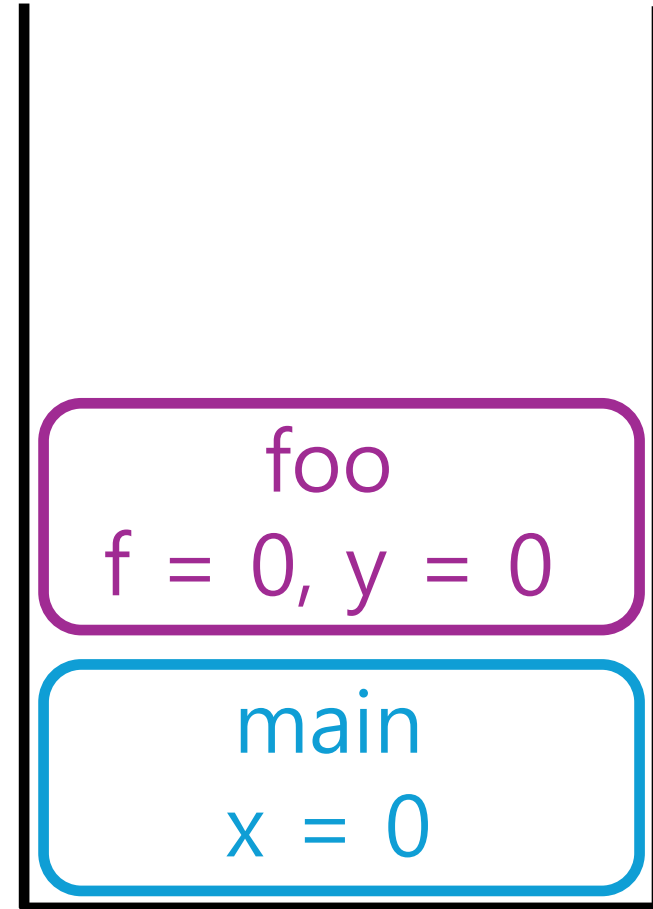
```
1  int main() {  
2      int x = 0;  
3      foo(x);  
4  }  
5  void foo(int f) {  
6      int y = f;  
7      bar(f);  
8  }  
9  void bar(int b) {  
10     int z = b;  
11 }
```



# Run-Time Stack



```
1  int main() {  
2      int x = 0;  
3      foo(x);  
4  }  
5  void foo(int f) {  
6      int y = f;  
7      bar(f);  
8  }  
9  void bar(int b) {  
10     int z = b;  
11 }
```

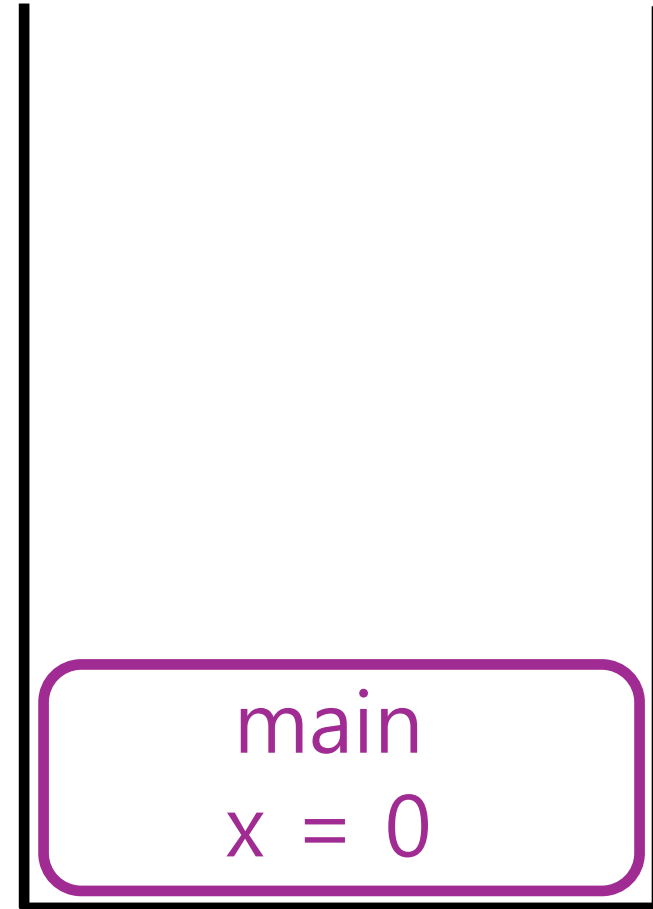




# Run-Time Stack



```
1  int main() {  
2      int x = 0;  
3      foo(x);  
4  }  
5  void foo(int f) {  
6      int y = f;  
7      bar(f);  
8  }  
9  void bar(int b) {  
10     int z = b;  
11 }
```



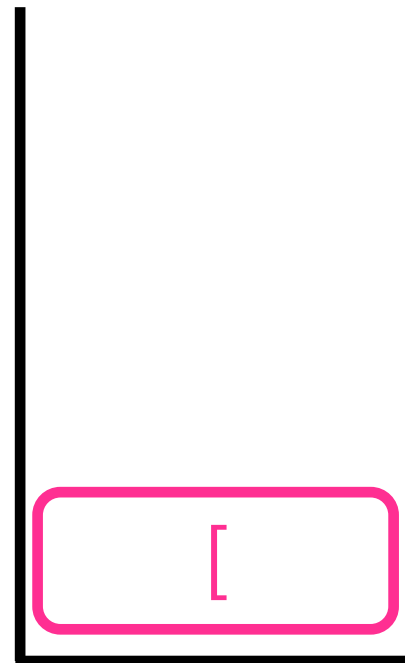
# Parentheses Matching

- 괄호 '(', '{', '['은 각각 ')', '}', ']'과 쌍이 맞게 이루어 져야 함
- 문장에서 **괄호의 쌍**이 맞는지 stack을 통해 점검
- **여는 괄호**를 만나면 stack에 **push**
- **닫는 괄호**를 만나면 stack에 top에 있는 괄호와 쌍이 맞으면 **pop**하고, 쌍이 맞지 않으면 matching 실패
- 문장을 모두 점검하고 stack에 **여는 괄호가 남아있지 않다**면 matching 성공

# Parentheses Matching

[ { 인생에서 중요한 점은  
(넘어짐)이 아닌 (일어섬)  
에 있다. }

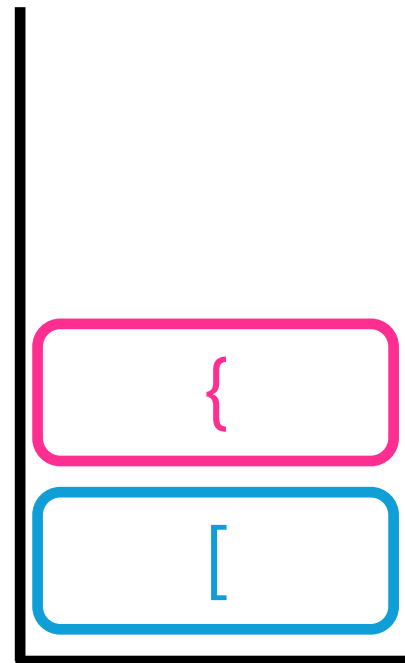
{ by JO. W. Y. }



# Parentheses Matching

[ { 인생에서 중요한 점은  
(넘어짐)이 아닌 (일어섬)  
에 있다. }

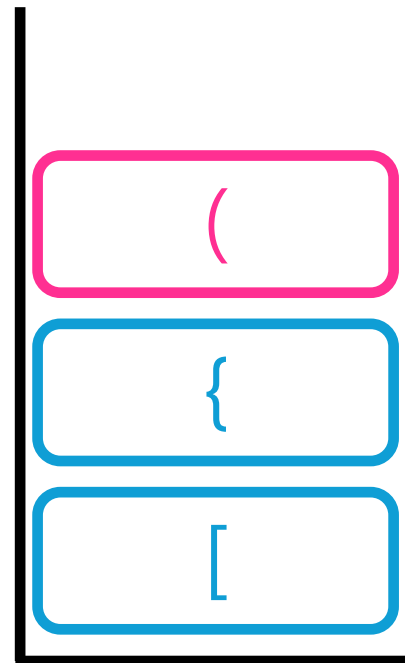
{ by JO. W. Y. }



# Parentheses Matching

[{ 인생에서 중요한 점은  
(넘어짐)이 아닌 (일어섬)  
에 있다. }

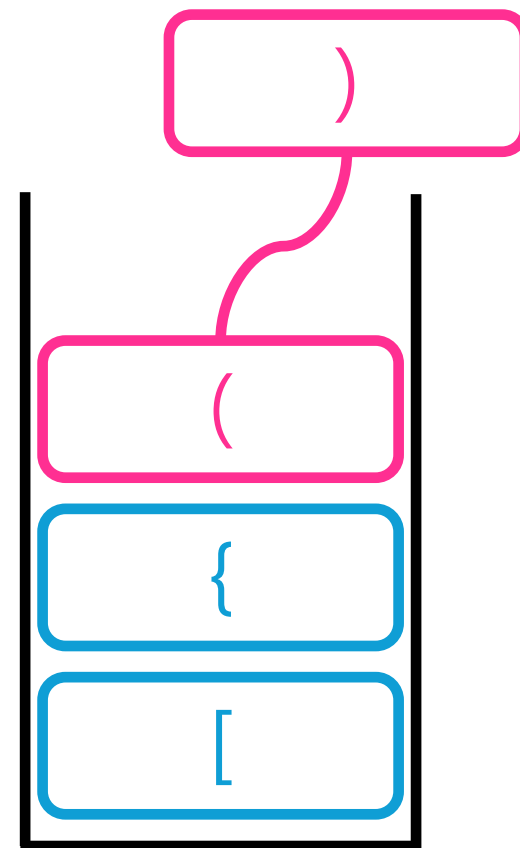
{ by JO. W. Y. }



# Parentheses Matching

[{ 인생에서 중요한 점은  
(넘어짐)이 아닌 (일어섬)  
에 있다. }

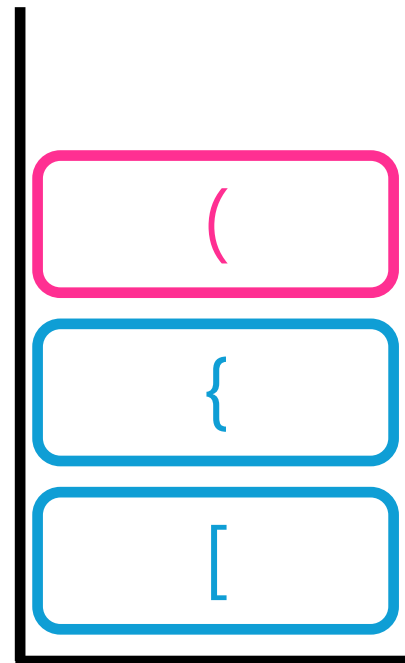
{ by JO. W. Y. }



# Parentheses Matching

[{ 인생에서 중요한 점은  
(넘어짐)이 아닌 (일어섬)  
에 있다. }

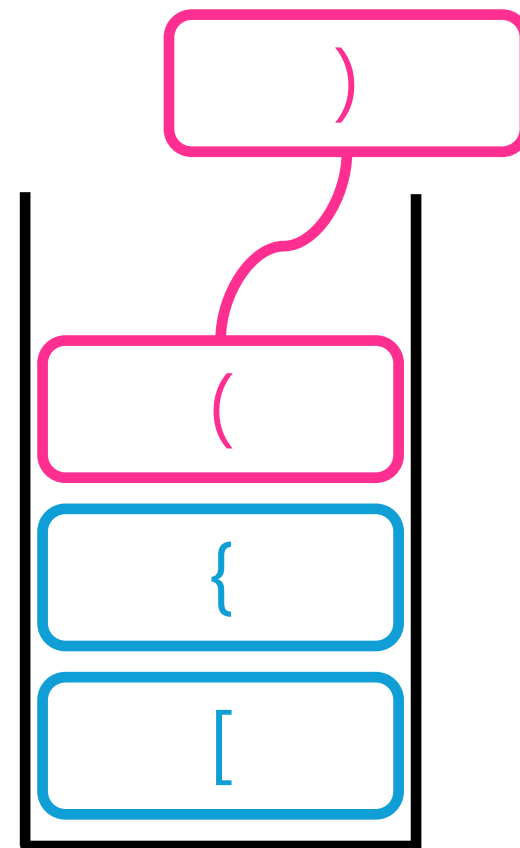
{ by JO. W. Y. }



# Parentheses Matching

[{ 인생에서 중요한 점은  
(넘어짐)이 아닌 (일어섬)  
에 있다. }

{ by JO. W. Y. }

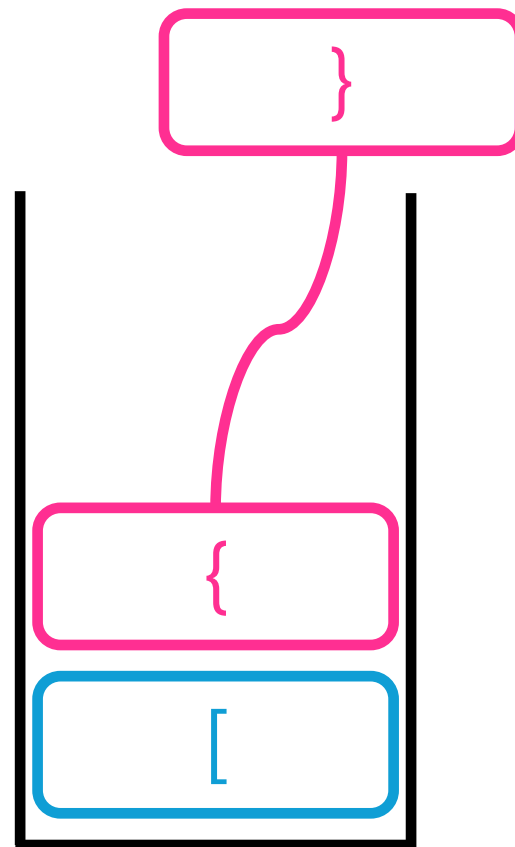




# Parentheses Matching

[{ 인생에서 중요한 점은  
(넘어짐)이 아닌 (일어섬)  
에 있다. }

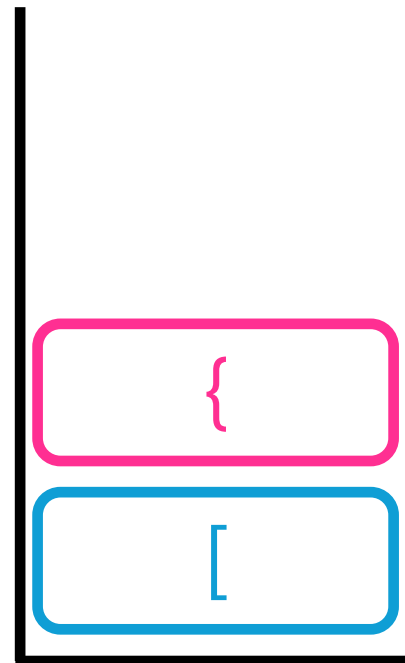
{ by JO. W. Y. }



# Parentheses Matching

[{ 인생에서 중요한 점은  
(넘어짐)이 아닌 (일어섬)  
에 있다. }

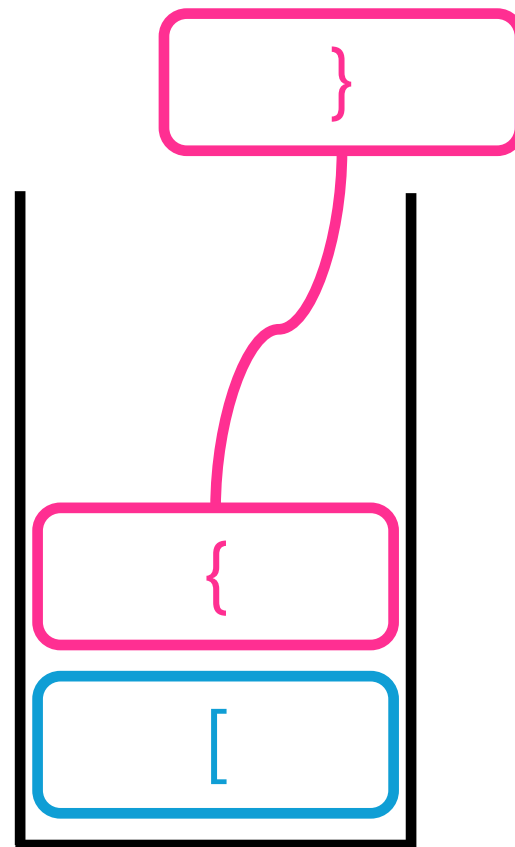
{ by JO. W. Y. }



# Parentheses Matching

[{ 인생에서 중요한 점은  
(넘어짐)이 아닌 (일어섬)  
에 있다. }

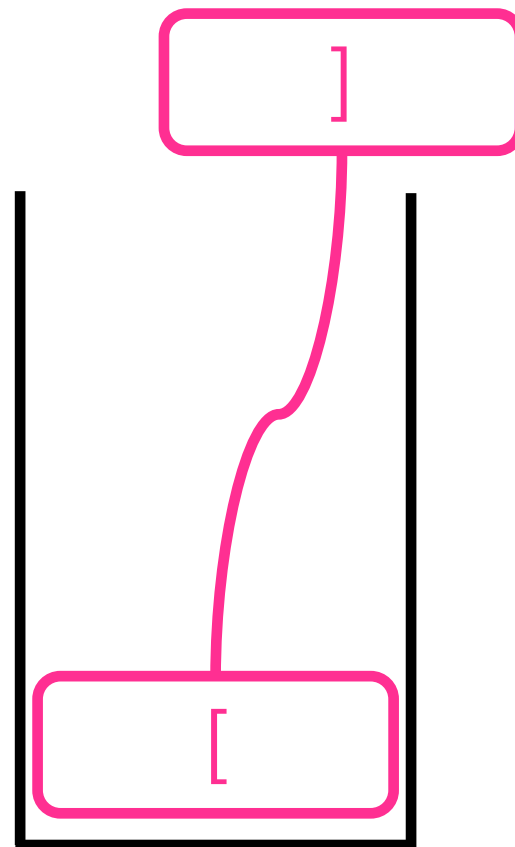
{ by JO. W. Y. }



# Parentheses Matching

[{ 인생에서 중요한 점은  
(넘어짐)이 아닌 (일어섬)  
에 있다. }

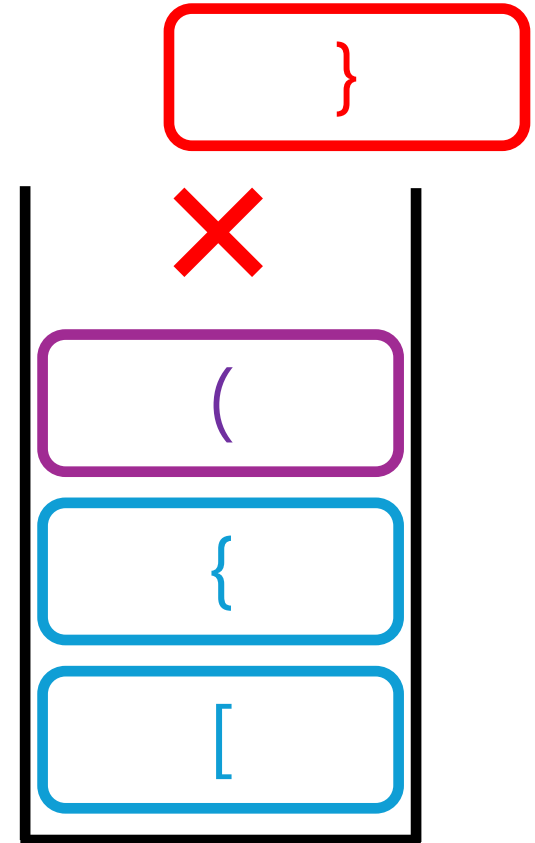
{ by JO. W. Y. }]



# Parentheses Matching Fail

[{ 인생에서 중요한 점은  
(넘어짐)이 아닌 (일어섬}  
에 있다. }

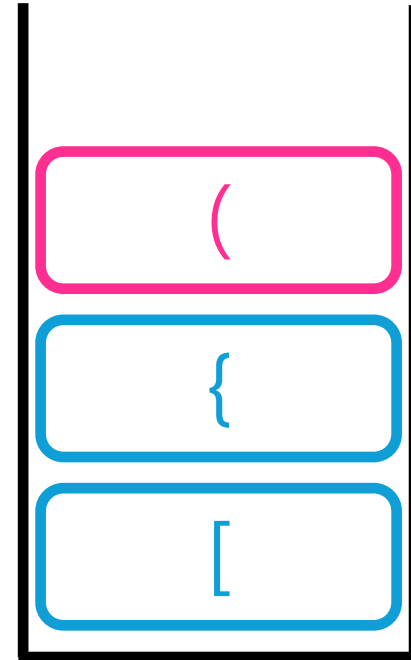
{ by JO. W. Y. }}



# Parentheses Matching Fail

[{ 인생에서 중요한 점은  
(넘어짐)이 (아닌 (일어섬)  
에 있다. }

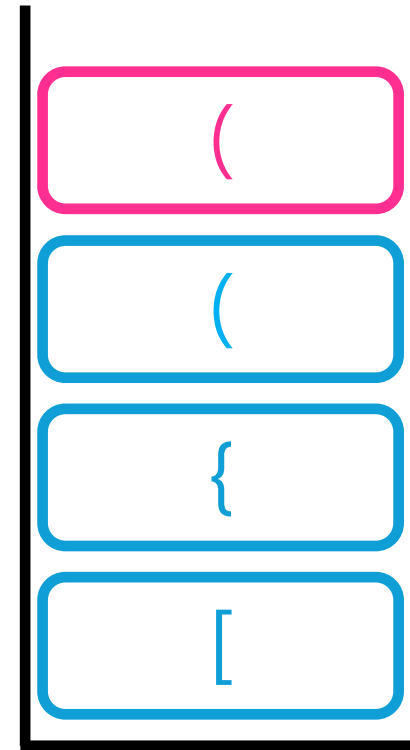
{ by JO. W. Y. }}



# Parentheses Matching Fail

[{ 인생에서 중요한 점은  
(넘어짐)이 (아닌 (일어섬)  
에 있다. }

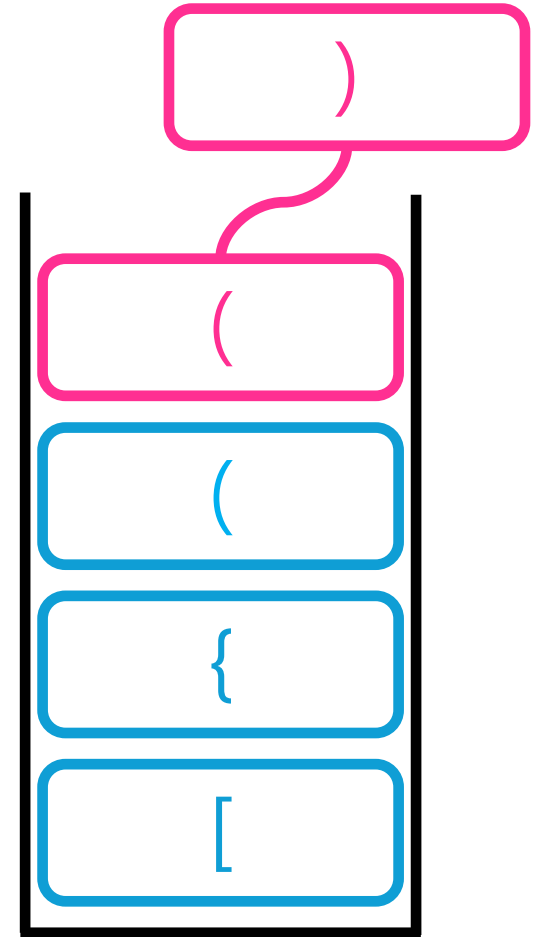
{ by JO. W. Y. }}



# Parentheses Matching Fail

[{ 인생에서 중요한 점은  
(넘어짐)이 (아닌 (일어섬)  
에 있다. }

{ by JO. W. Y. }}

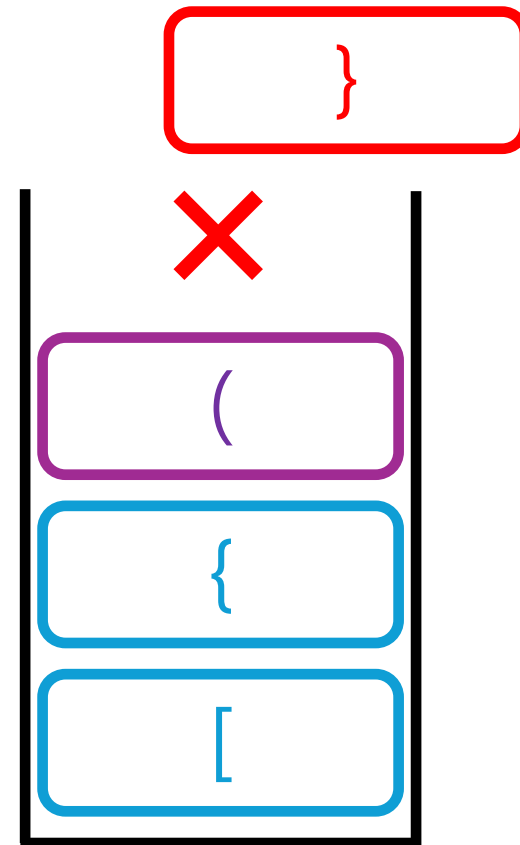




# Parentheses Matching Fail

[{ 인생에서 중요한 점은  
(넘어짐)이 (아닌 (일어섬)  
에 있다. }

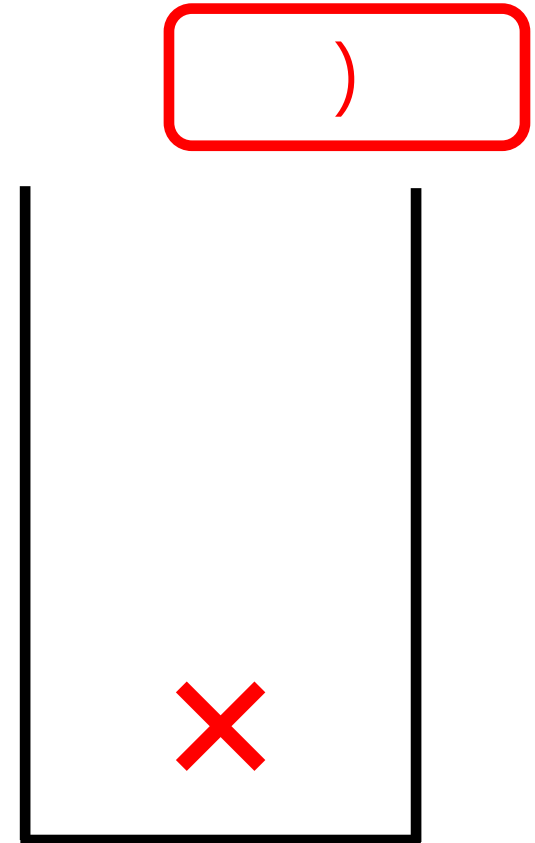
{ by JO. W. Y. }}



# Parentheses Matching Fail

)[{ 인생에서 중요한 점은  
(넘어짐)이 아닌 (일어섬)  
에 있다. }

{ by JO. W. Y. }



# Parentheses Matching Code



```
1  bool parentheses_matching(string str) {  
2      ArrayStack2<char> stack;  
3  
4      char open[3] = {'[', '{', '('};  
5      char close[3] = {']', '}', ')'};  
6      for (int i = 0; i < str.length(); i++) {  
7          if (!is_parentheses(str[i]))  
8              continue;
```

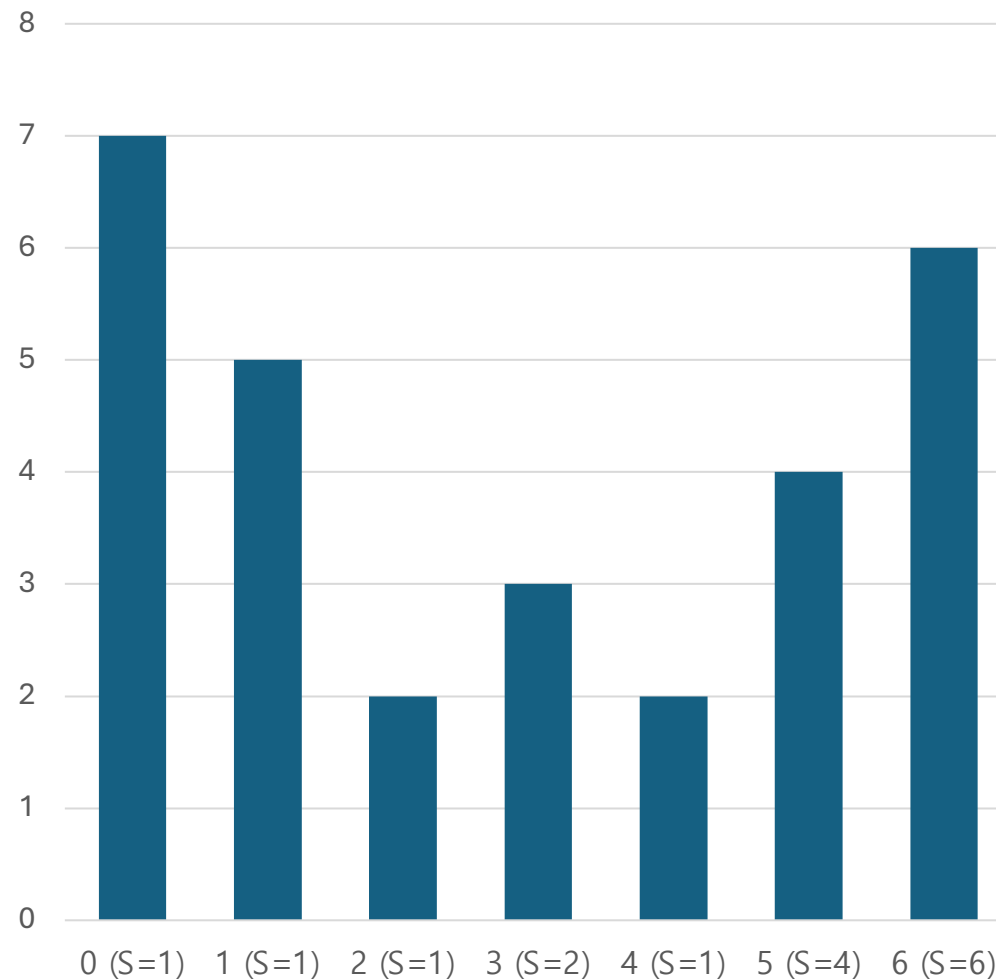
# Parentheses Matching Code



```
1      for (int j = 0; j < 3; j++) {  
2          if (str[i] == close[j] && !stack.empty() && stack.top() == open[j]) {  
3              stack.pop();  
4              break;  
5          }  
6          else if (j == 2) {  
7              stack.push(str[i]);  
8          }  
9      }  
10 }  
11  
12 return stack.empty();  
13 }
```

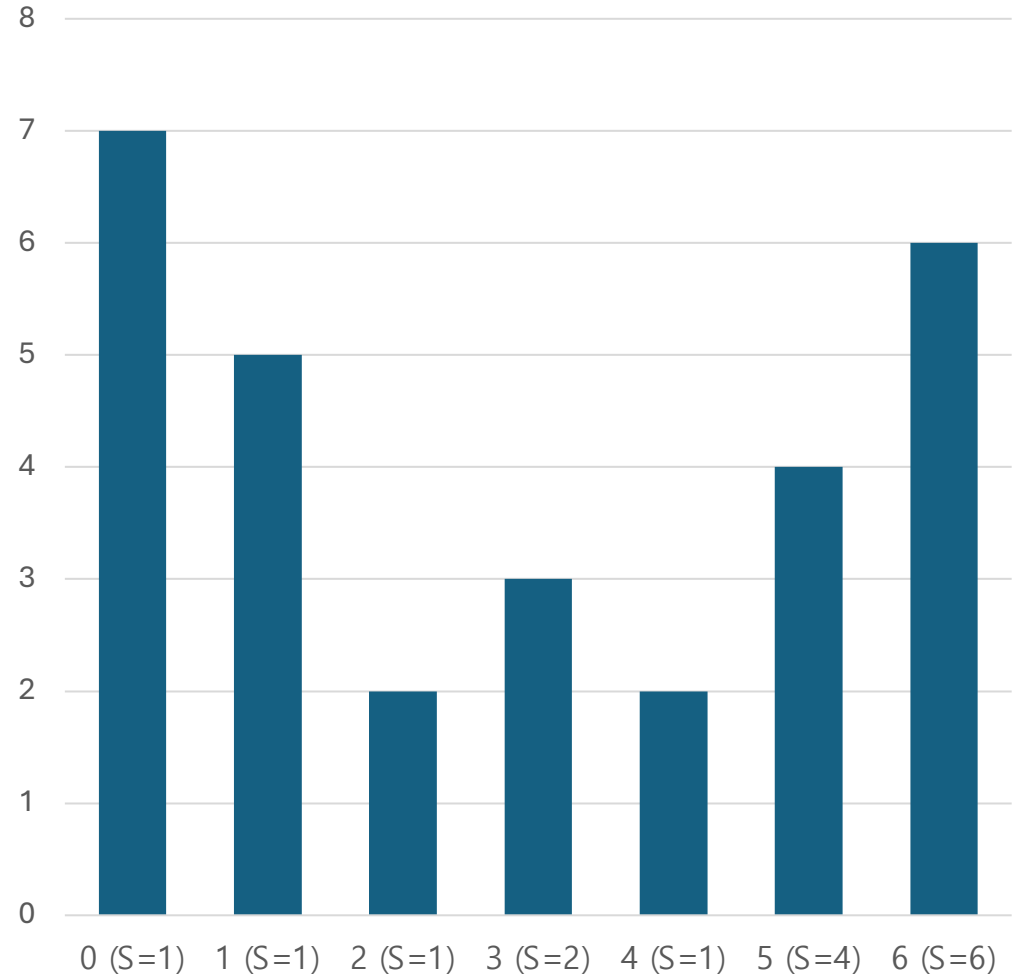
# Stock Span Problem

- 일별로 주식의 가격이 주어지고, 일별로 **주식 가격의 span**을 구하는 문제
- $i$ 일에 대한 span  $s_i$ 는  $i$ 일 직전의 연속된 최대 일수이며, 연속될 수 있음의 조건은  $i$ 일의 주식 가격보다 작거나 같아야 함



# Quadratic Algorithm

- 현재 날짜보다 이전 날짜들의 주식 가격을 탐색
- 이전 날짜에서 현재 날짜보다 주식 가격이 낮다면  $s_i := s_i + 1$  을 하고 주식 가격을 계속 탐색
- 이전 날짜의 주식 가격이 현재 보다 높다면 탐색 종료



# Quadratic Algorithm Code



```
1  int *quadratic_span(int *price, int len) {
2      int *span = new int[len];
3
4      for (int i = 0; i < len; i++) {
5          span[i] = 1;
6          for (int j = i - 1; j >= 0 && price[j] <= price[i]; j--) {
7              span[i]++;
8          }
9      }
10
11     return span;
12 }
```

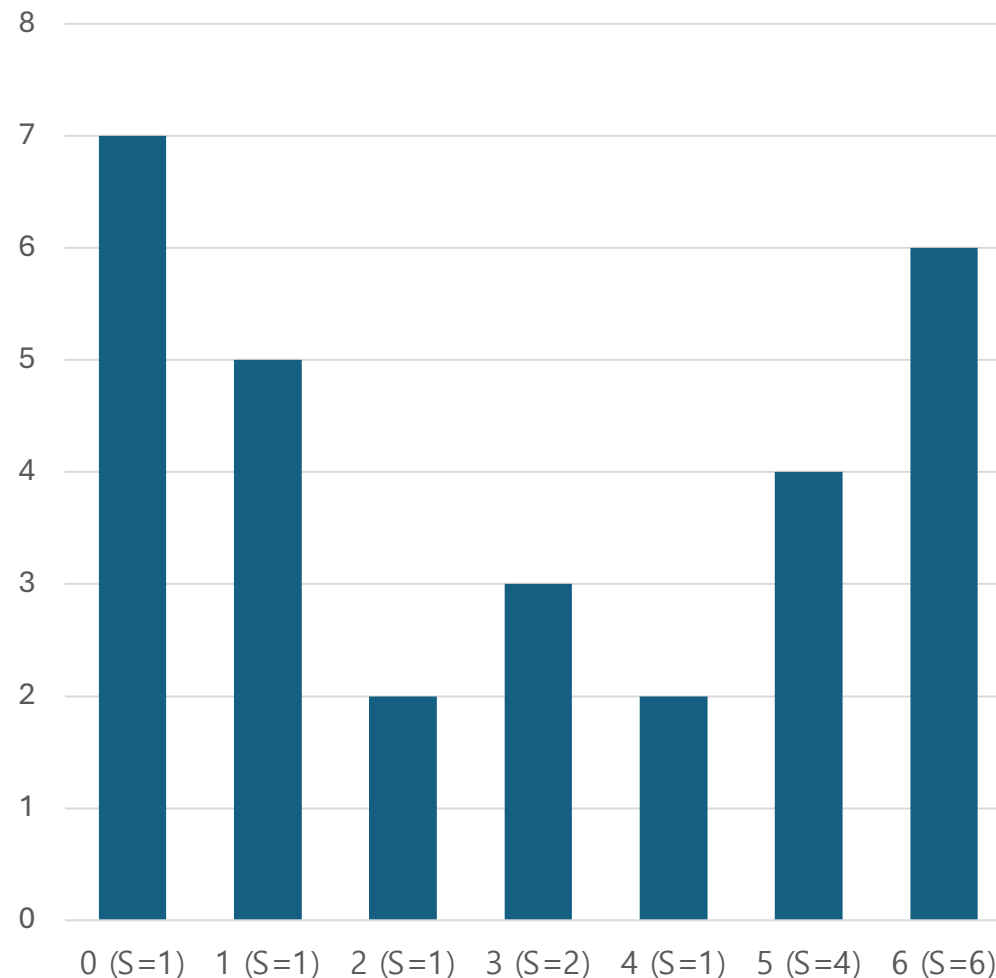
# Quadratic Algorithm 분석

- 단위연산 : 이전 날짜의 가격과 현재 날짜의 **가격 비교**
- 입력크기 : 날짜의 길이 **len**
- 날짜별로 span을 계산하는 바깥 for문은 **len번 실행**
- span을 구하기 위해 안쪽 for문의 **최악의 경우는 주식 가격이 오름  
차순이 되어 마지막 날짜에 모든 일자에 대해 가격을 비교**하여 **len  
번 실행**
- **$W(n) = n \times n \in O(n^2)$**



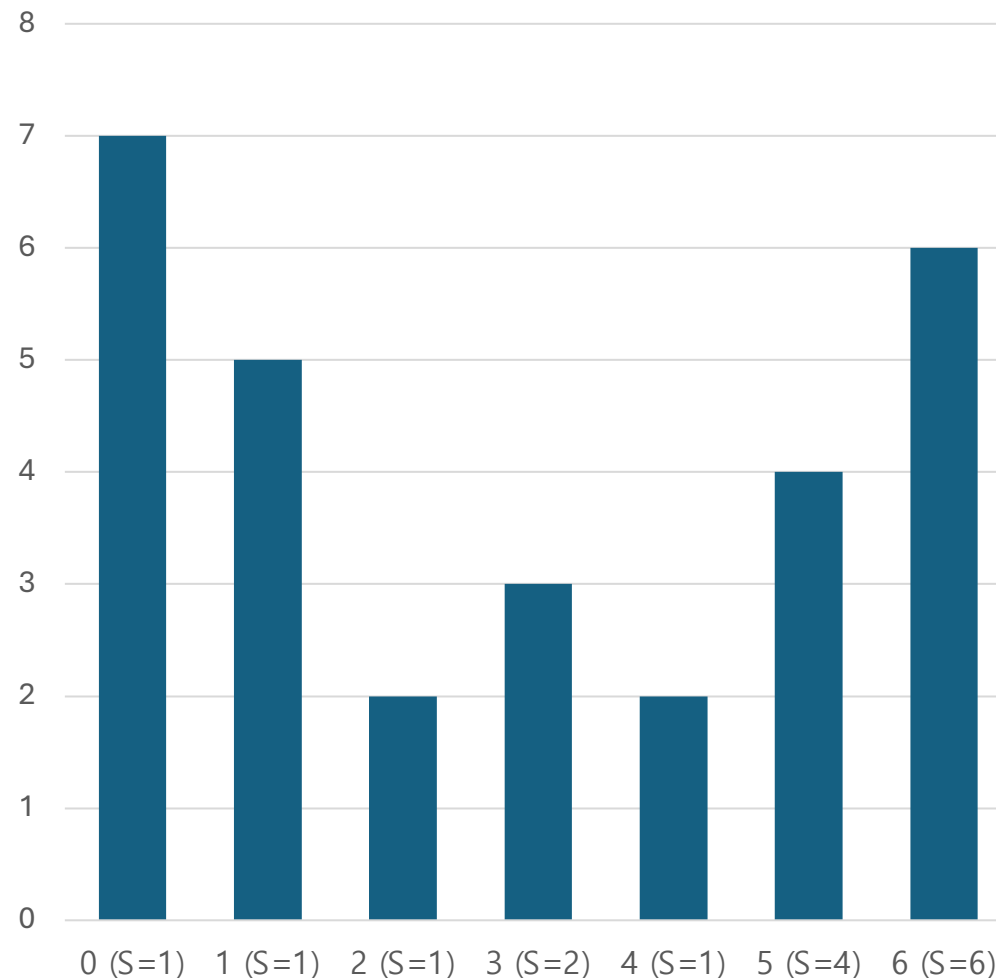
# Linear Algorithm

- **stack**을 사용해 **선형시간**에 span을 계산
- 일자별로 span을 계산하고 **현재 날짜를 push**
- span을 계산할 때 stack의 마지막 원소가 **현재 날짜의 원소보다 큰 가격의 날짜** 나오기 전까지 **pop**

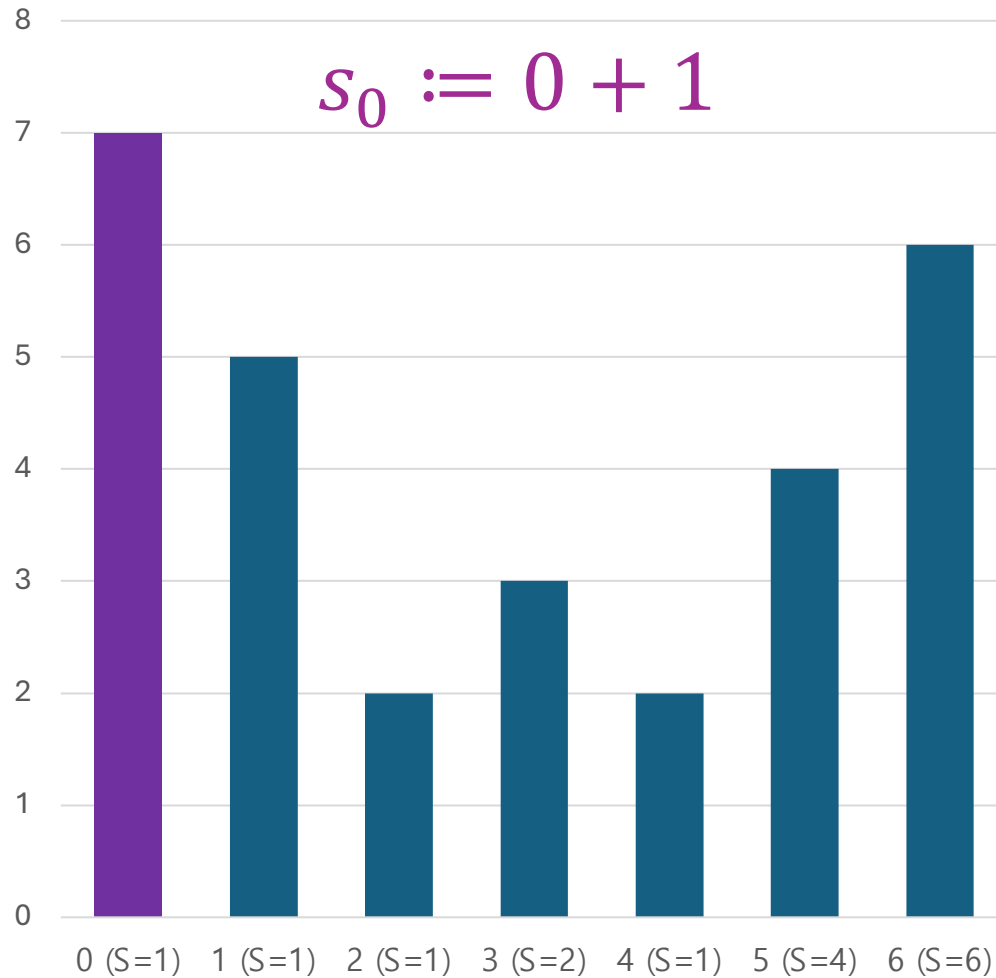


# Linear Algorithm

- $i$ 일에 대해 pop을 완료하고  
stack이 비어 있다면  $s_i := i + 1$ 
  - stack이 비어 있다는 의미는  $i$ 이  
전에  $i$ 일보다 높은 가격이 없음을  
의미
- stack에 원소가 있다면  $s_i :=$   
 $i - \text{stack.top}$ 
  - stack에 원소가 있다는 의미는  $i$   
이전에 가장 가격이 높은 날짜는  
 $\text{stack.top}$ 에 해당하는 날

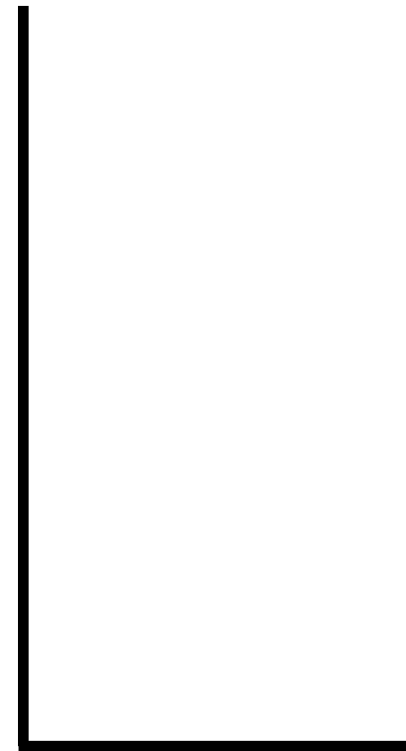


# Linear Algorithm

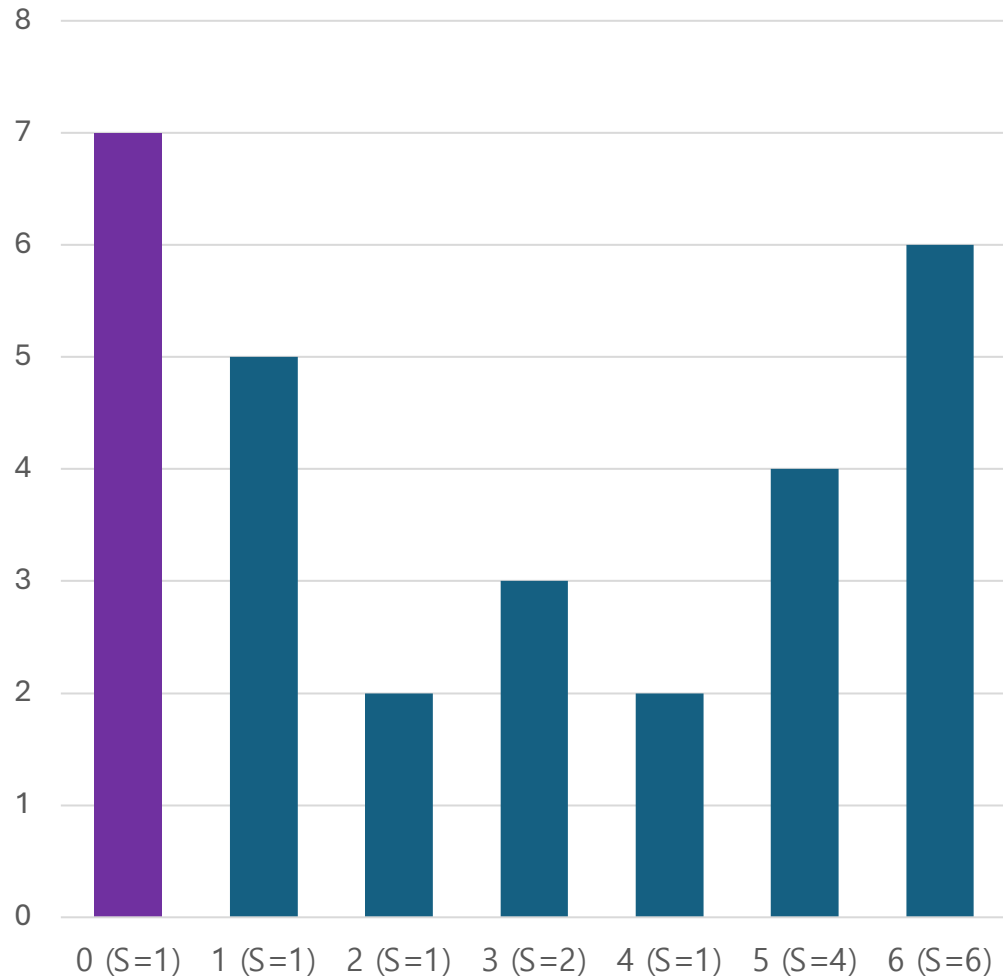


$$s_0 := 0 + 1$$

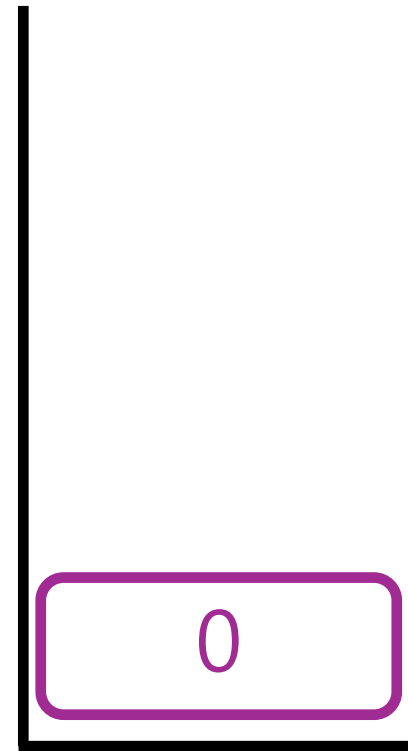
$$s_i := i + 1$$



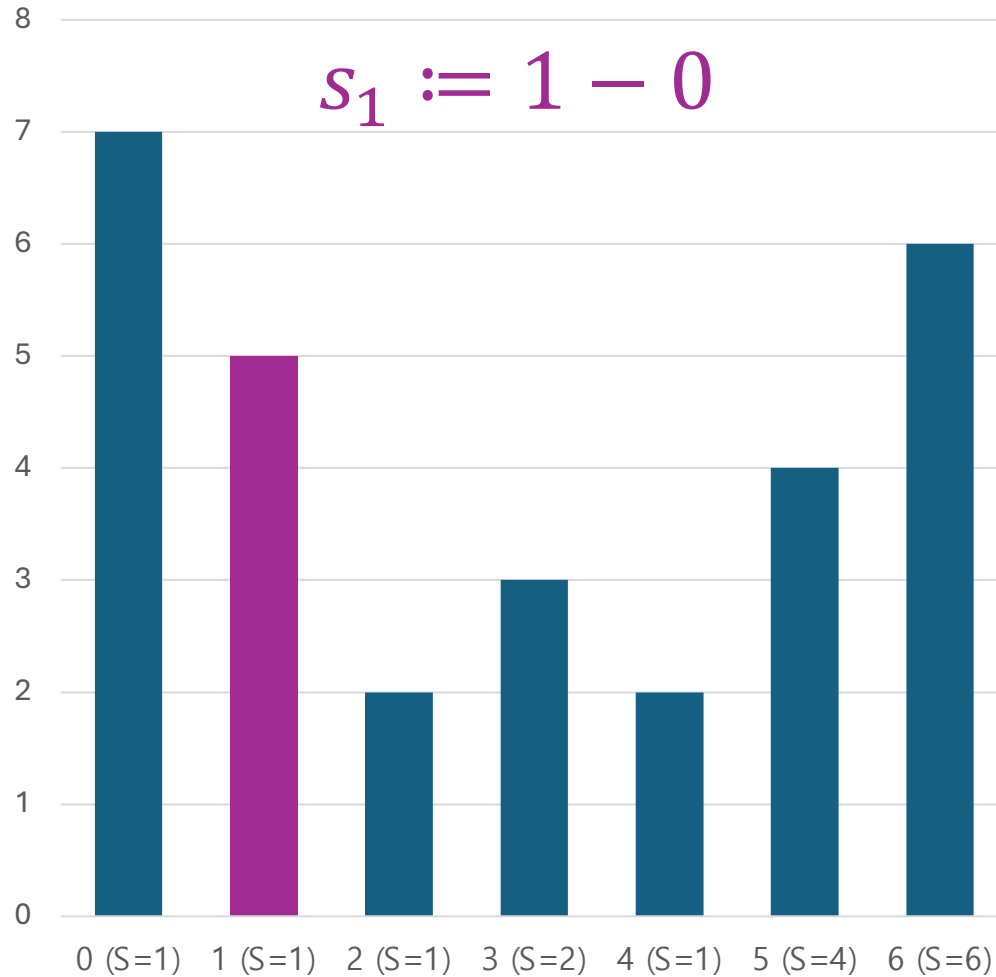
# Linear Algorithm



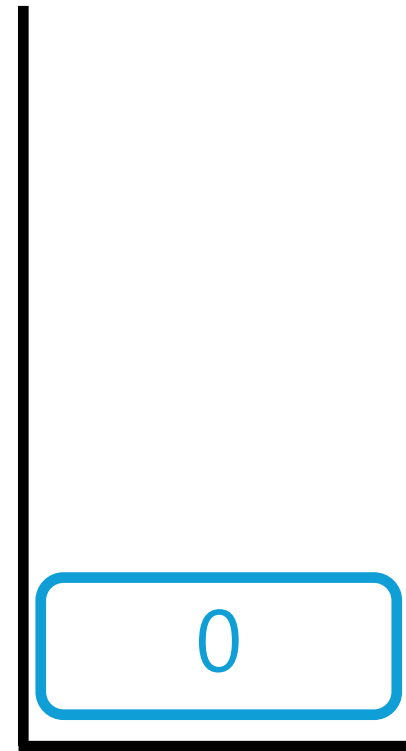
$$s_i := i + 1$$



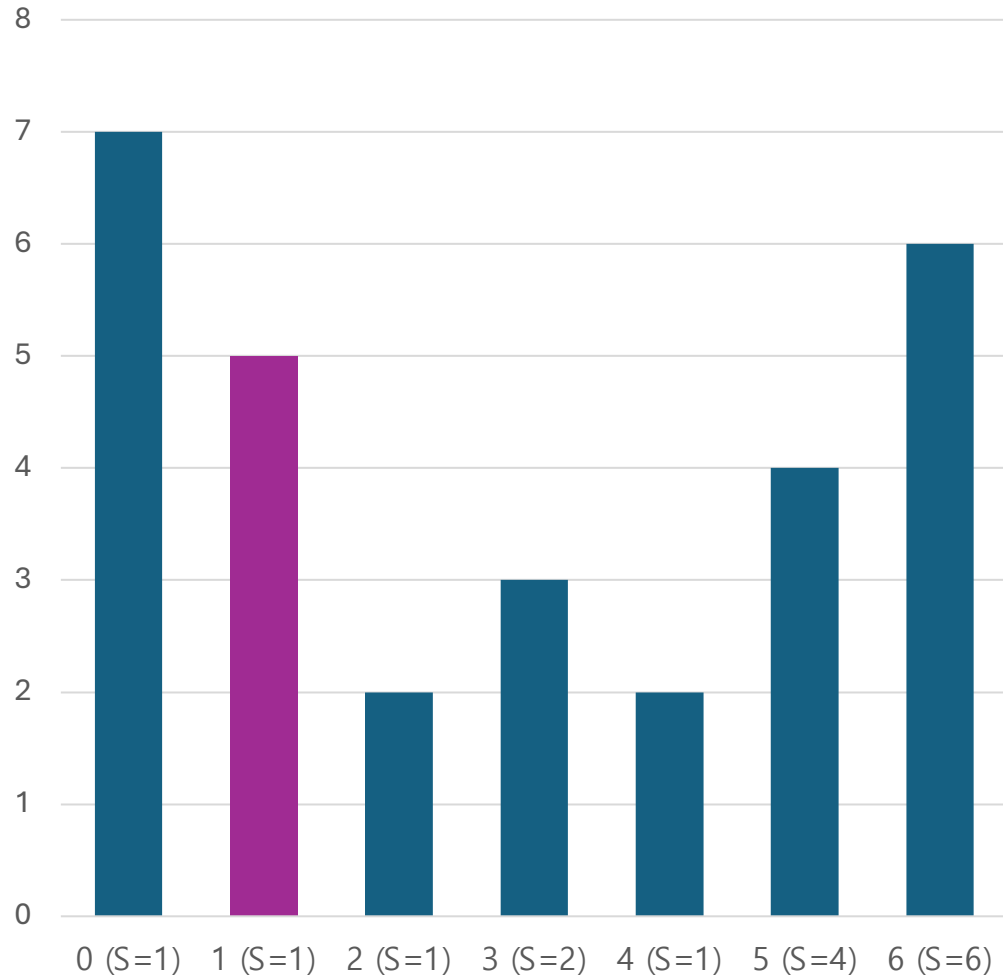
# Linear Algorithm



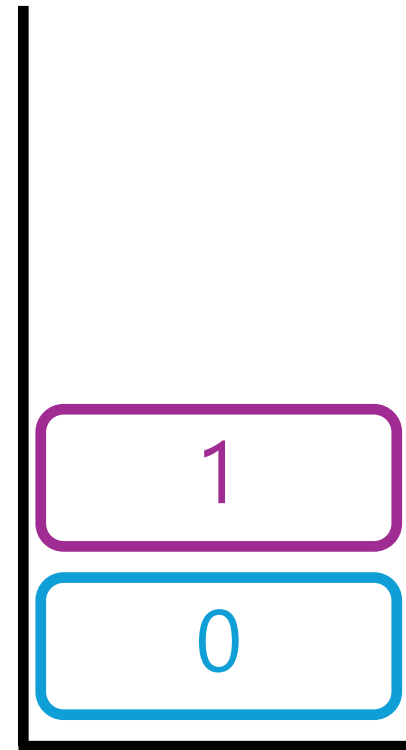
$$s_i := i - \text{stack.top}$$



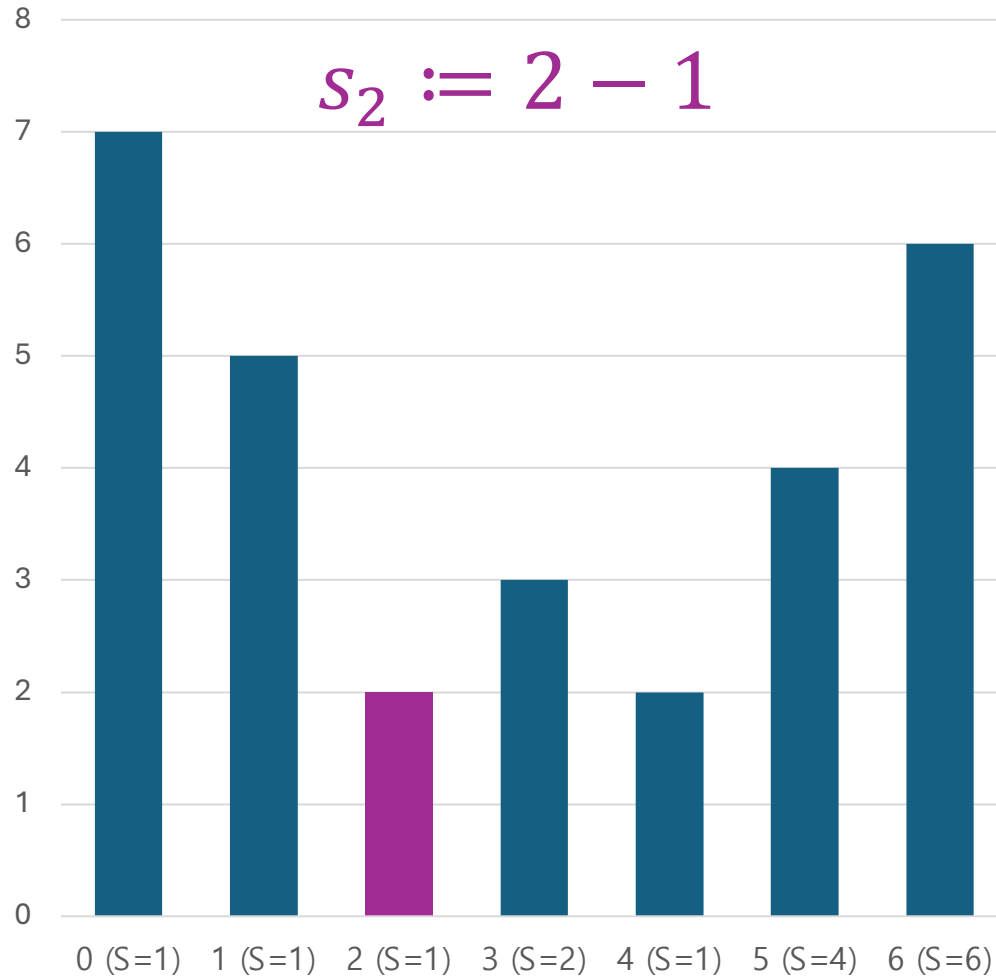
# Linear Algorithm



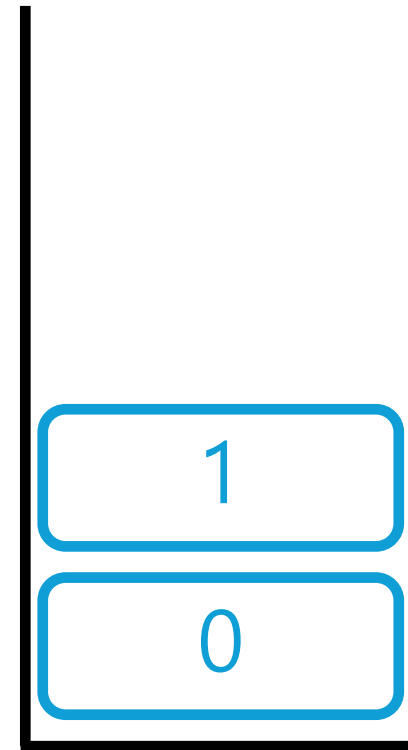
$$s_i := i - \text{stack.top}$$



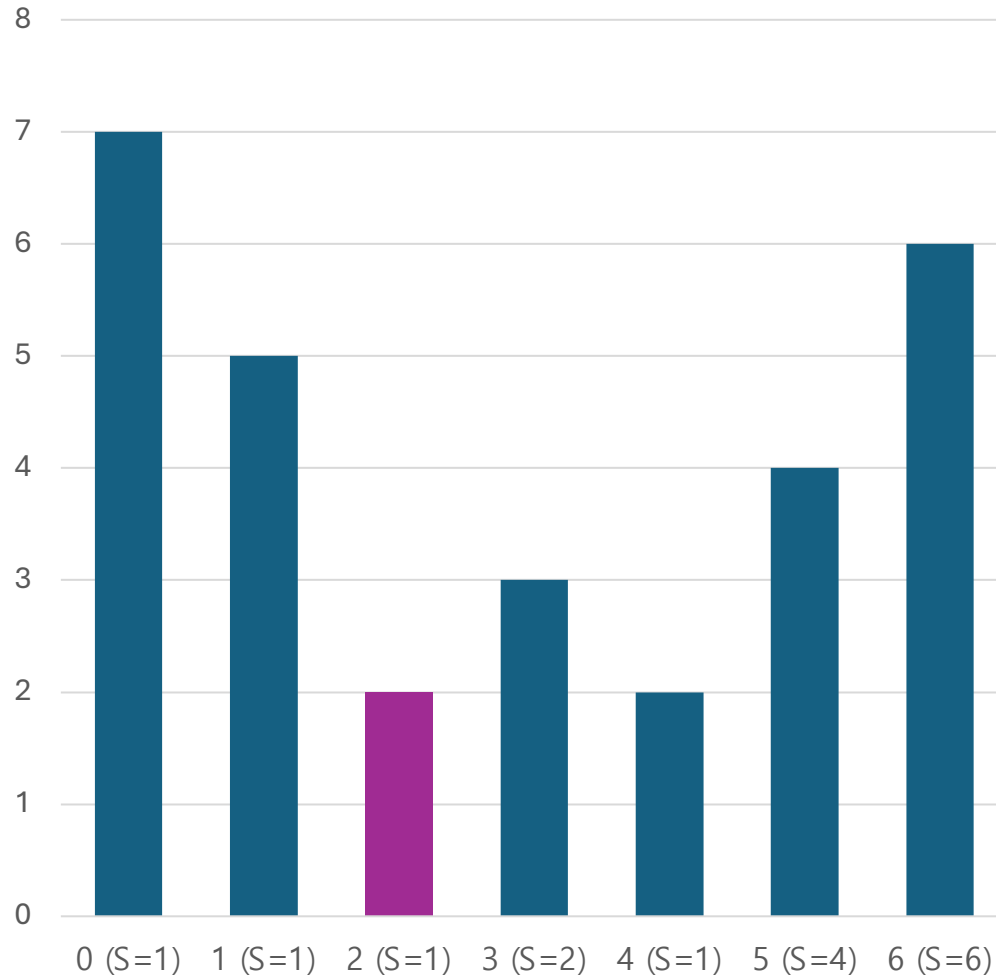
# Linear Algorithm



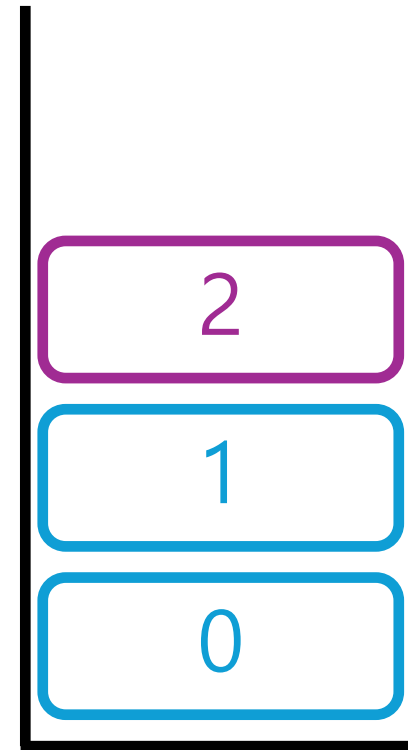
$$s_i := i - \text{stack.top}$$



# Linear Algorithm

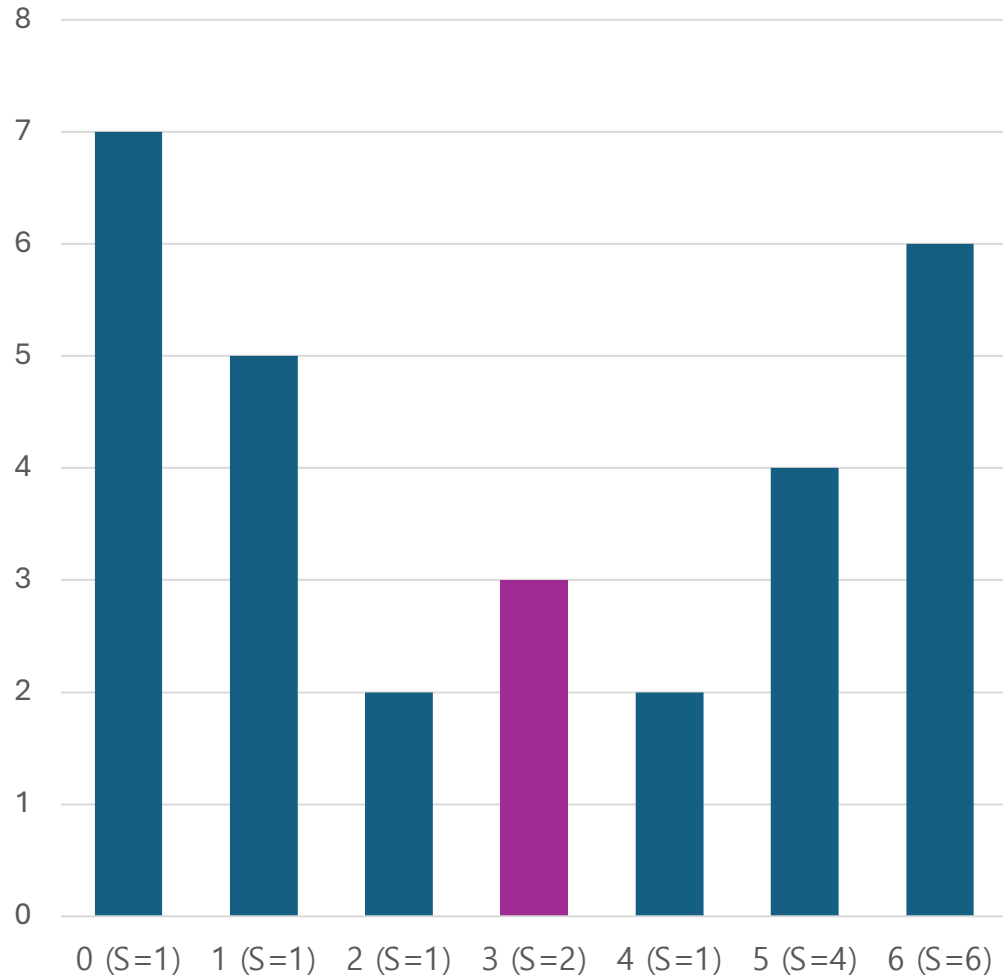


$$s_i := i - \text{stack.top}$$

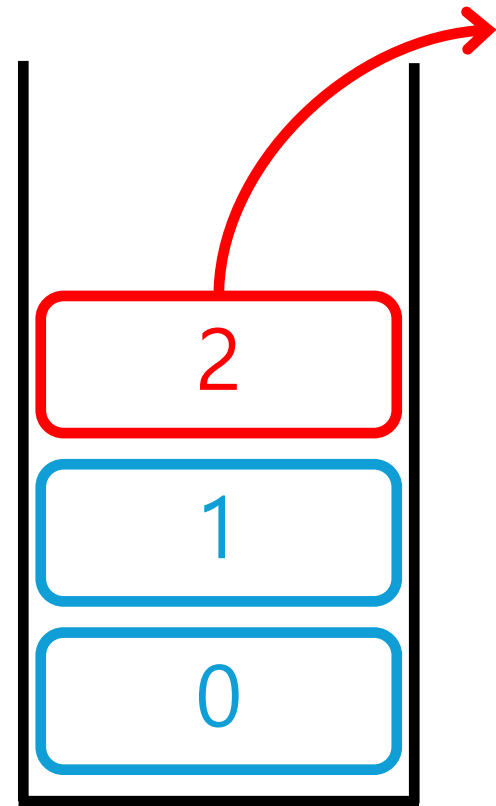




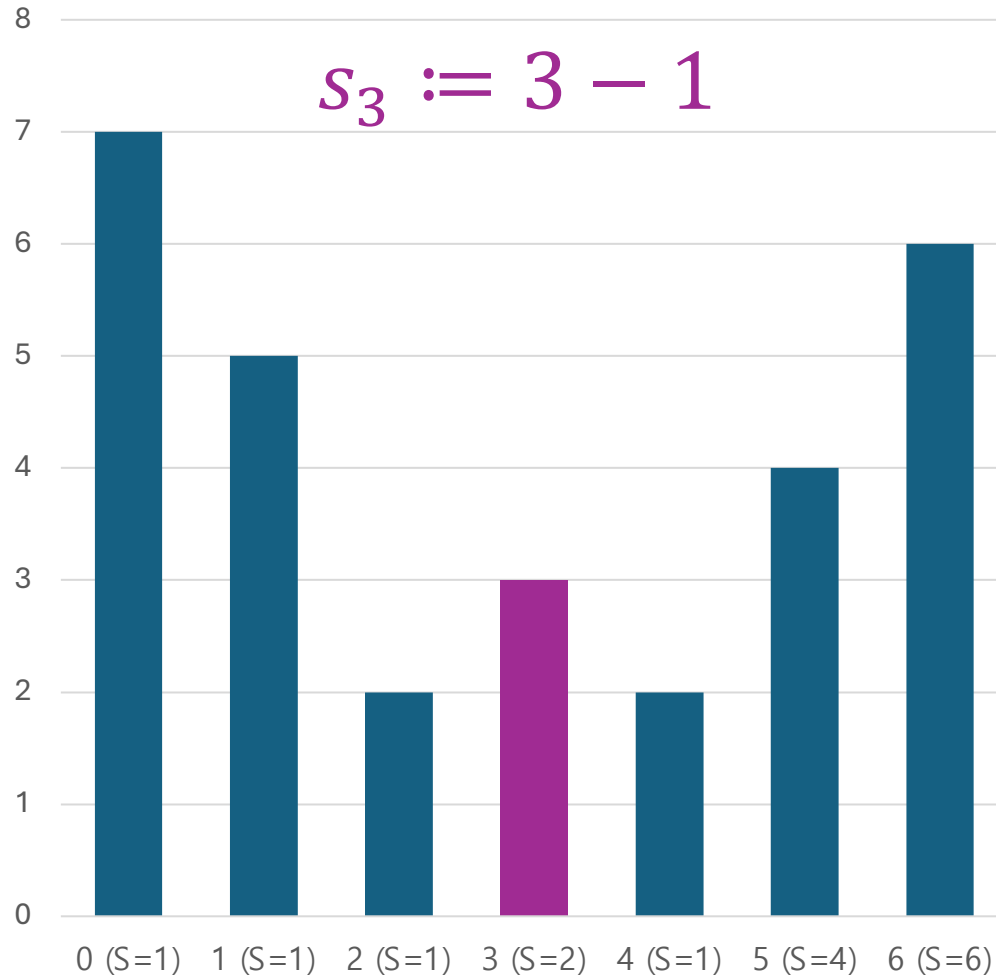
# Linear Algorithm



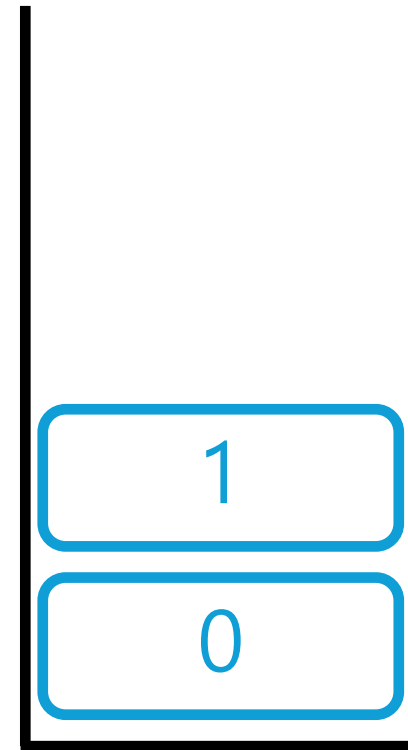
$$s_i := i - \text{stack.top}$$



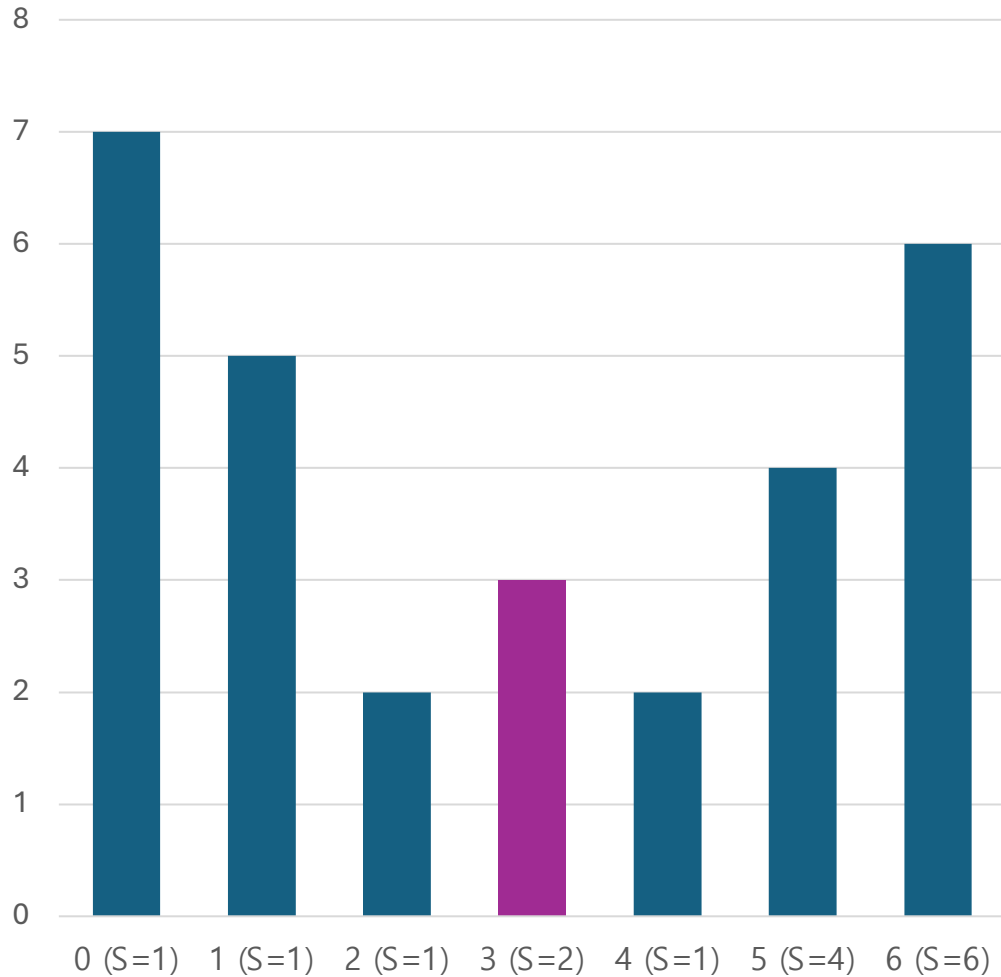
# Linear Algorithm



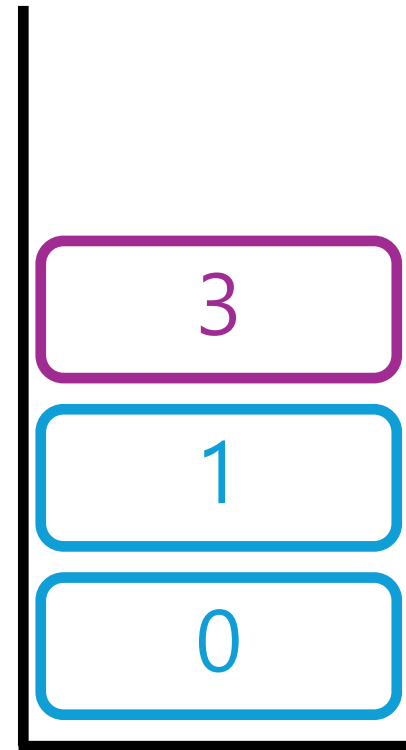
$$s_i := i - stack.top$$



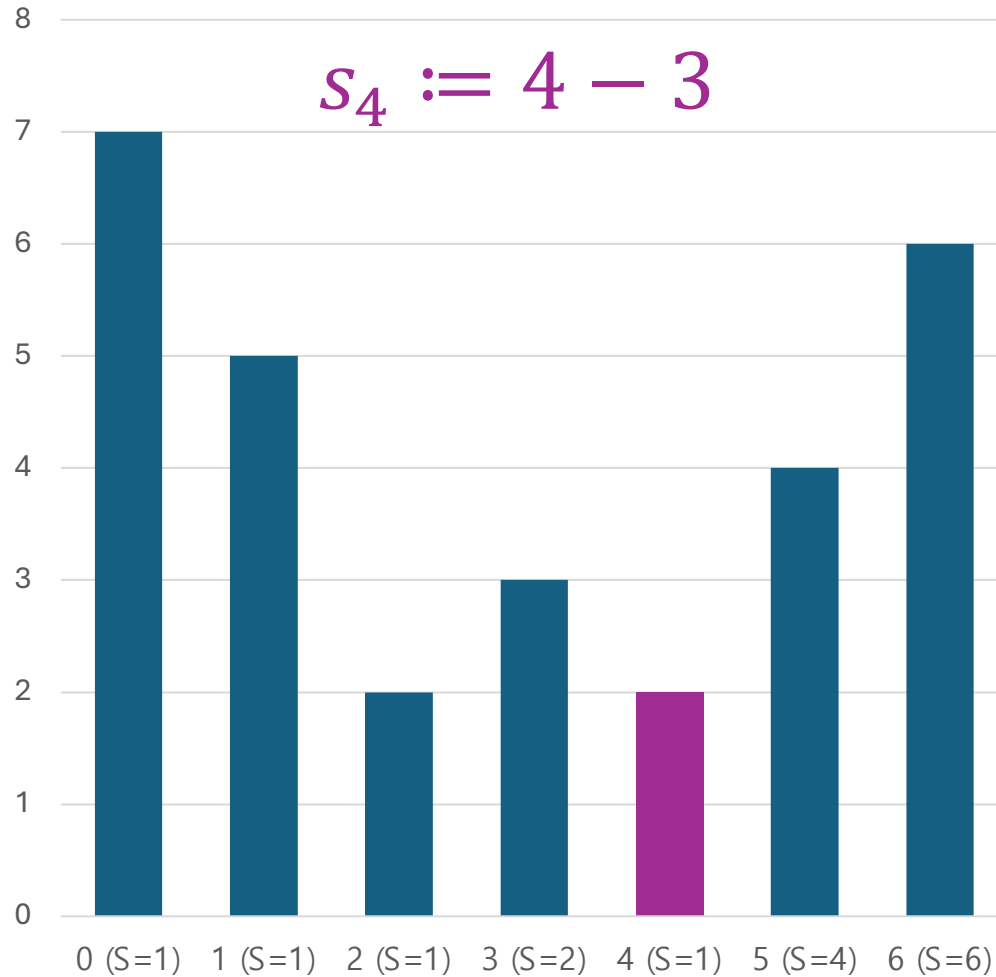
# Linear Algorithm



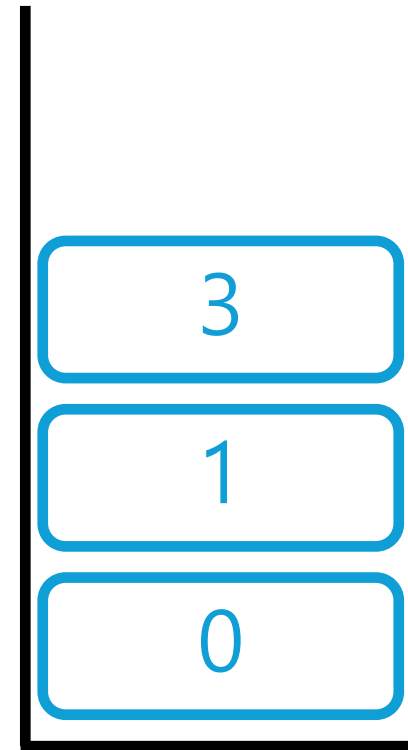
$$s_i := i - \text{stack.top}$$



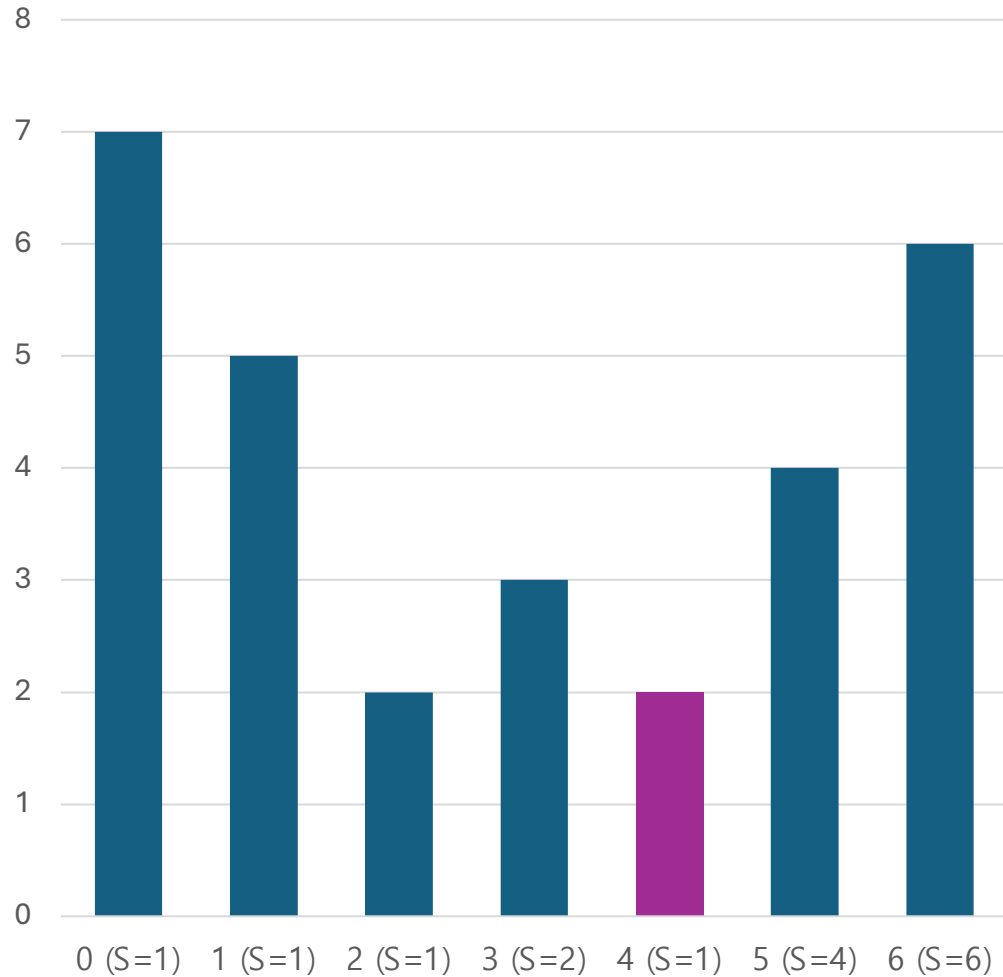
# Linear Algorithm



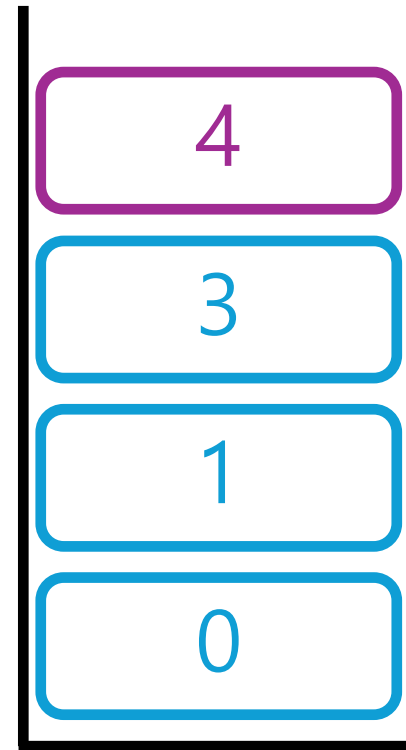
$$s_i := i - \text{stack.top}$$



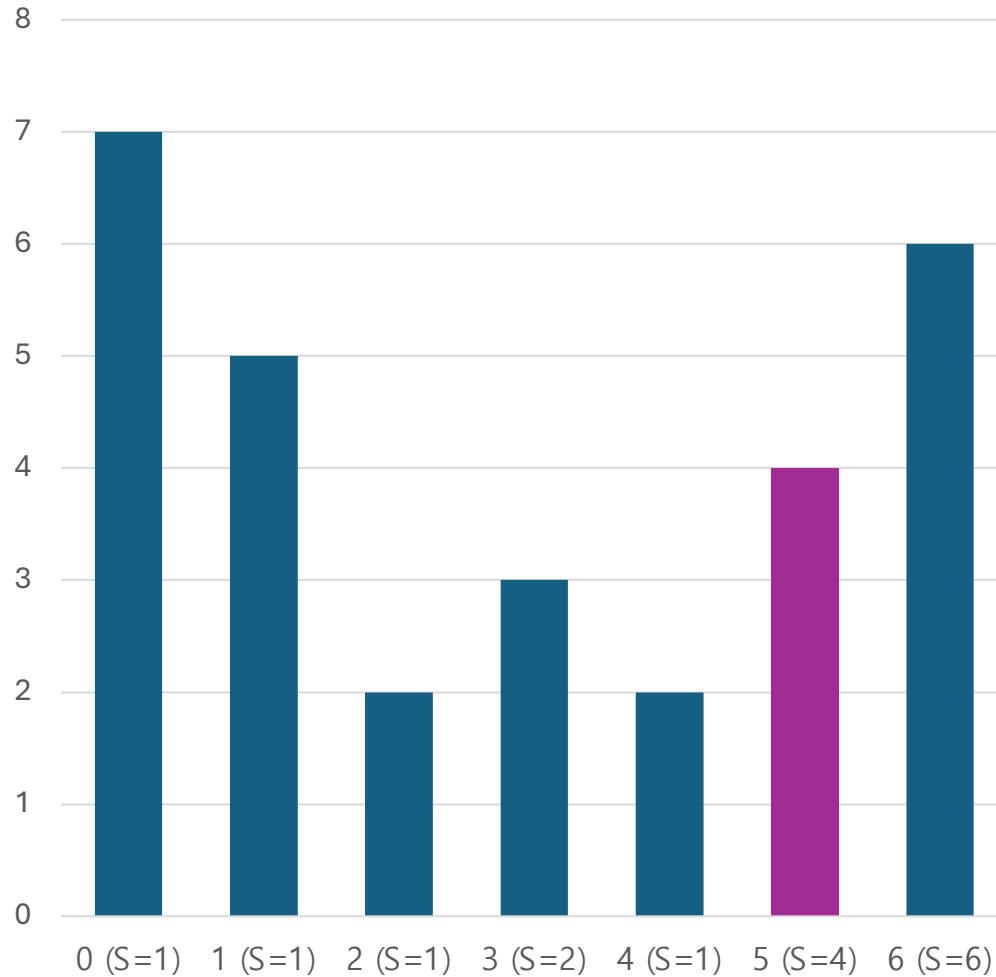
# Linear Algorithm



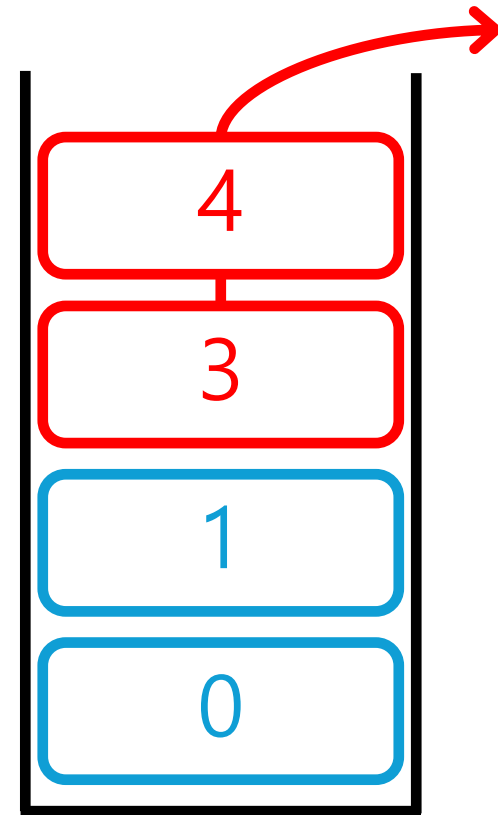
$$s_i := i - \text{stack.top}$$



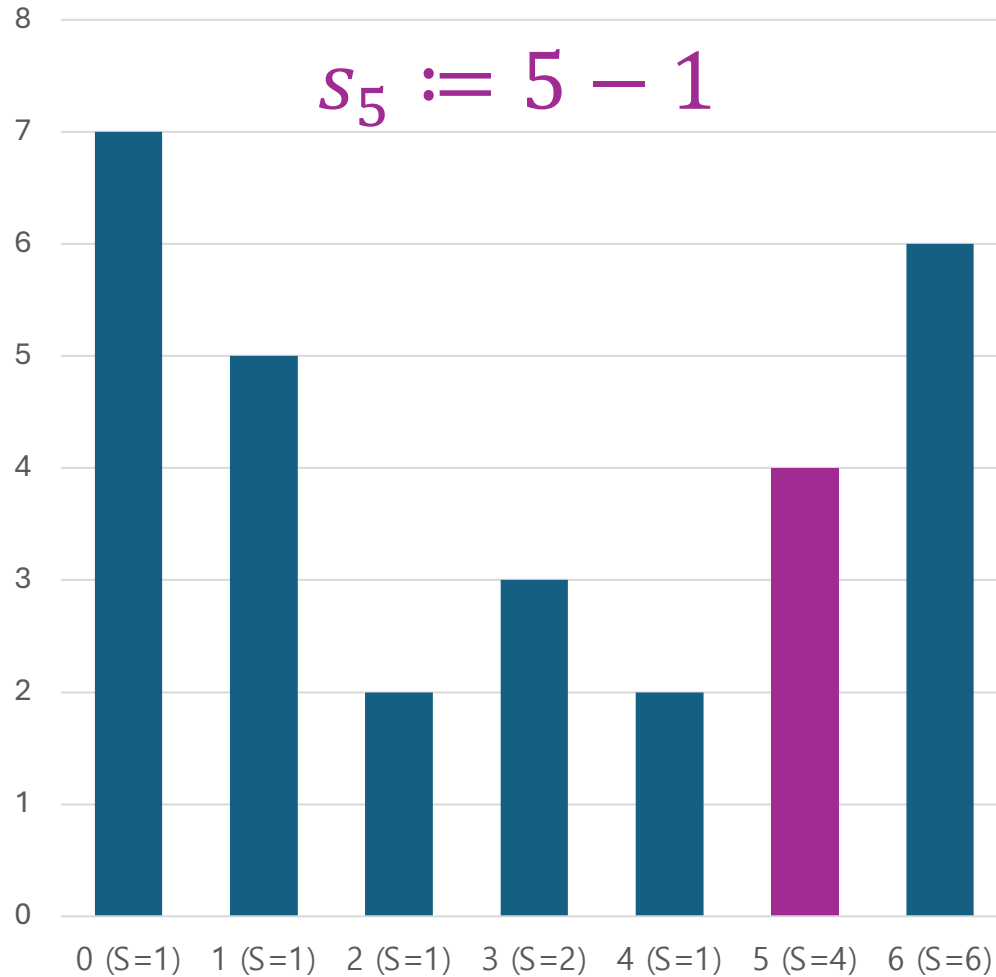
# Linear Algorithm



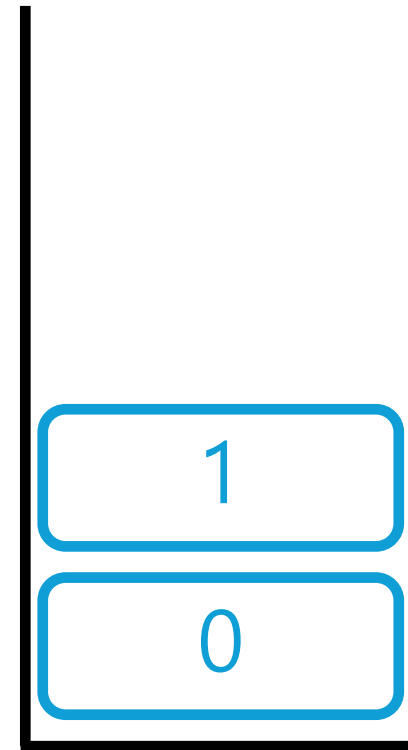
$$s_i := i - \text{stack.top}$$



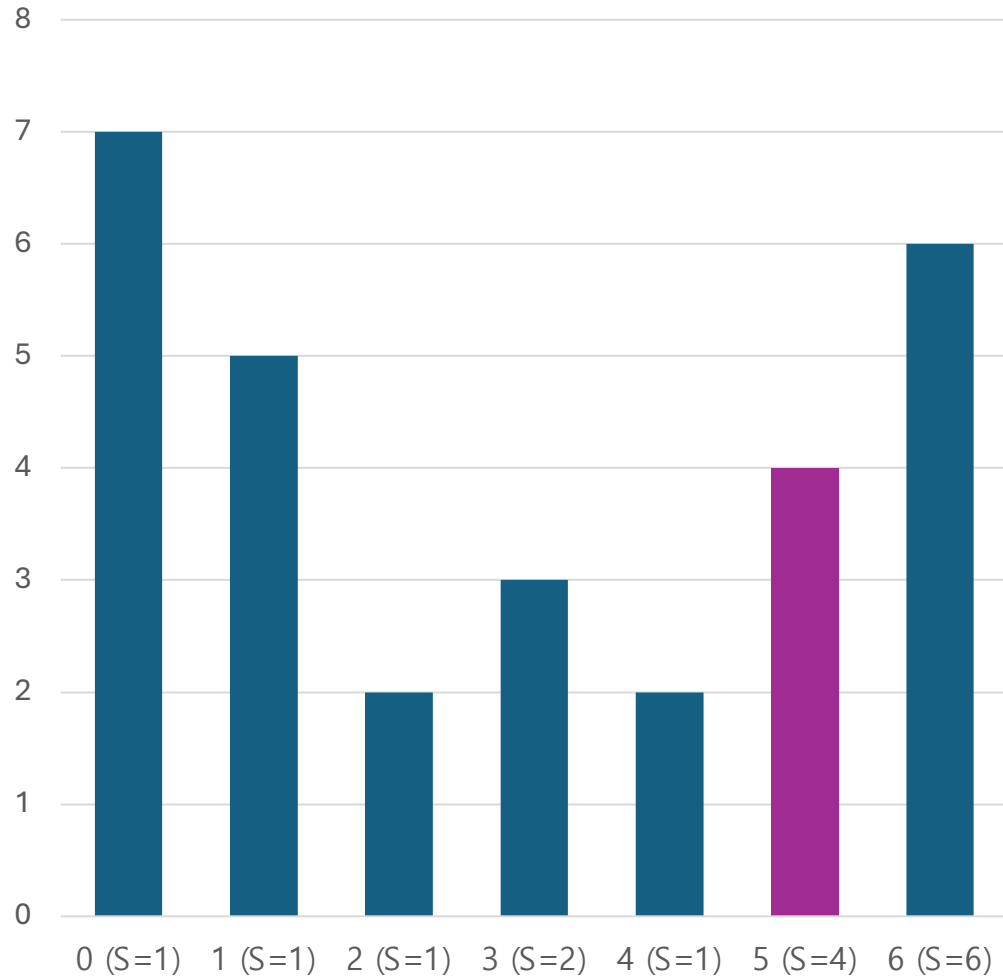
# Linear Algorithm



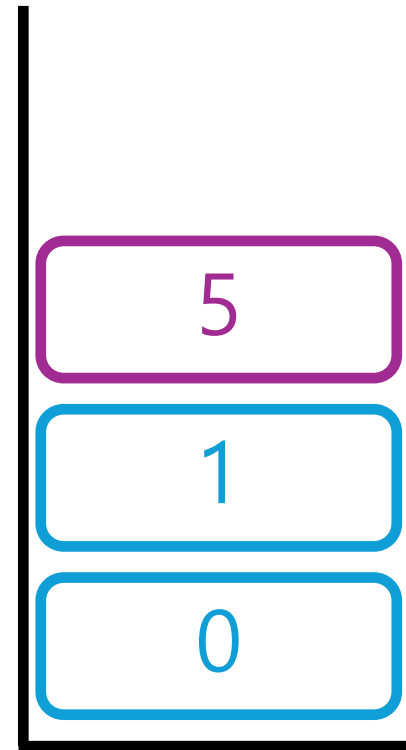
$$s_i := i - stack.top$$



# Linear Algorithm

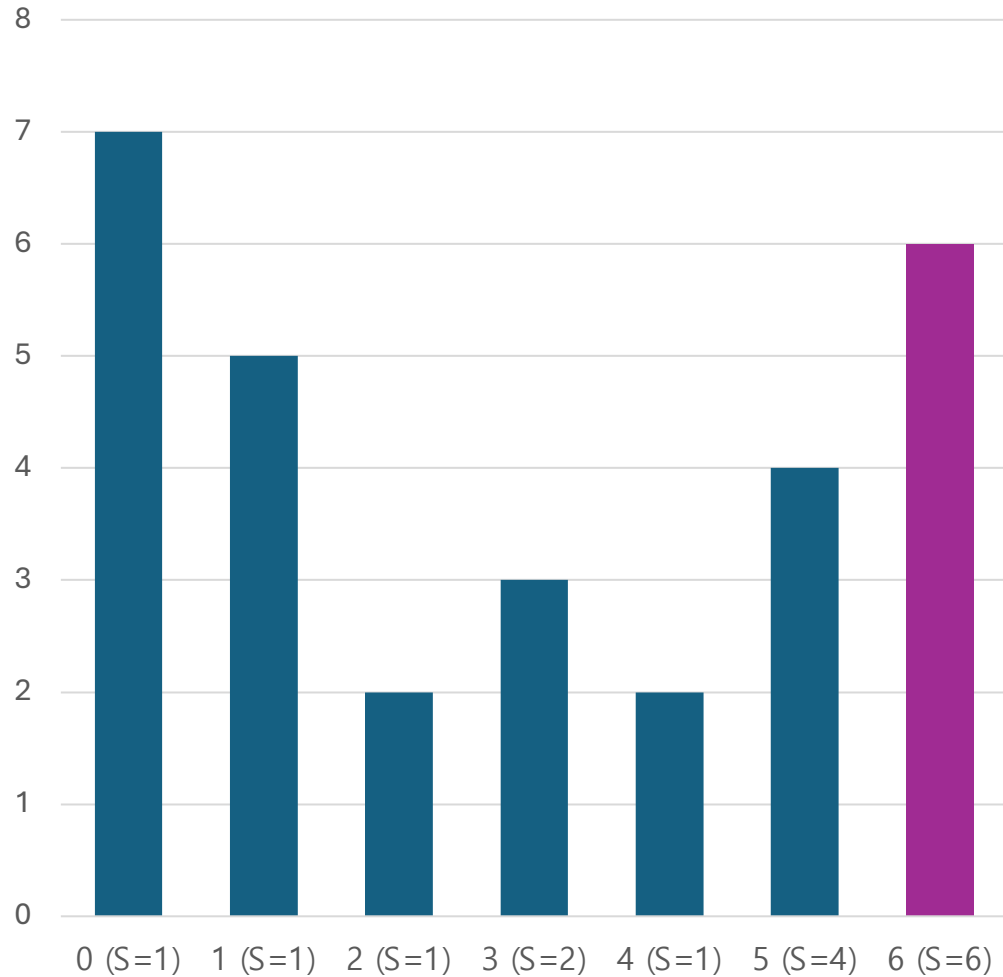


$$s_i := i - \text{stack.top}$$

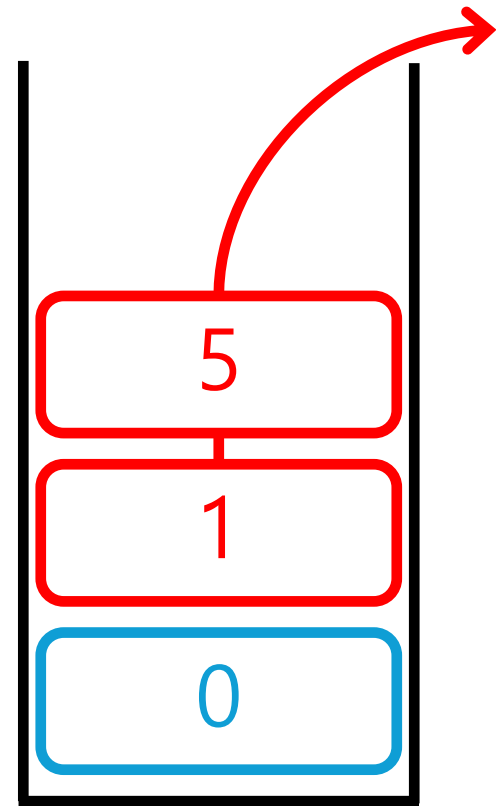




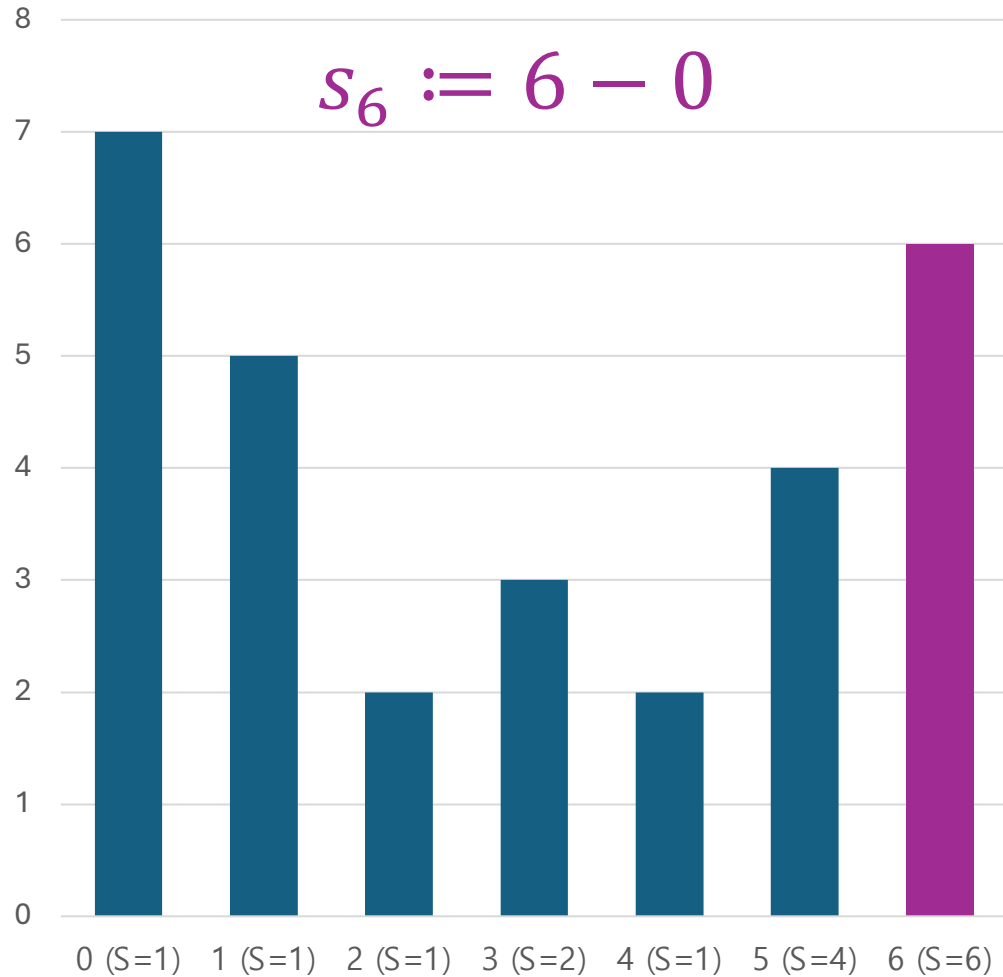
# Linear Algorithm



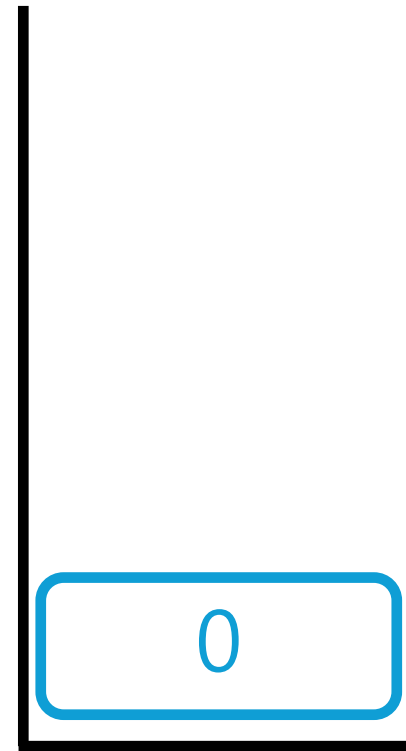
$$s_i := i - \text{stack.top}$$



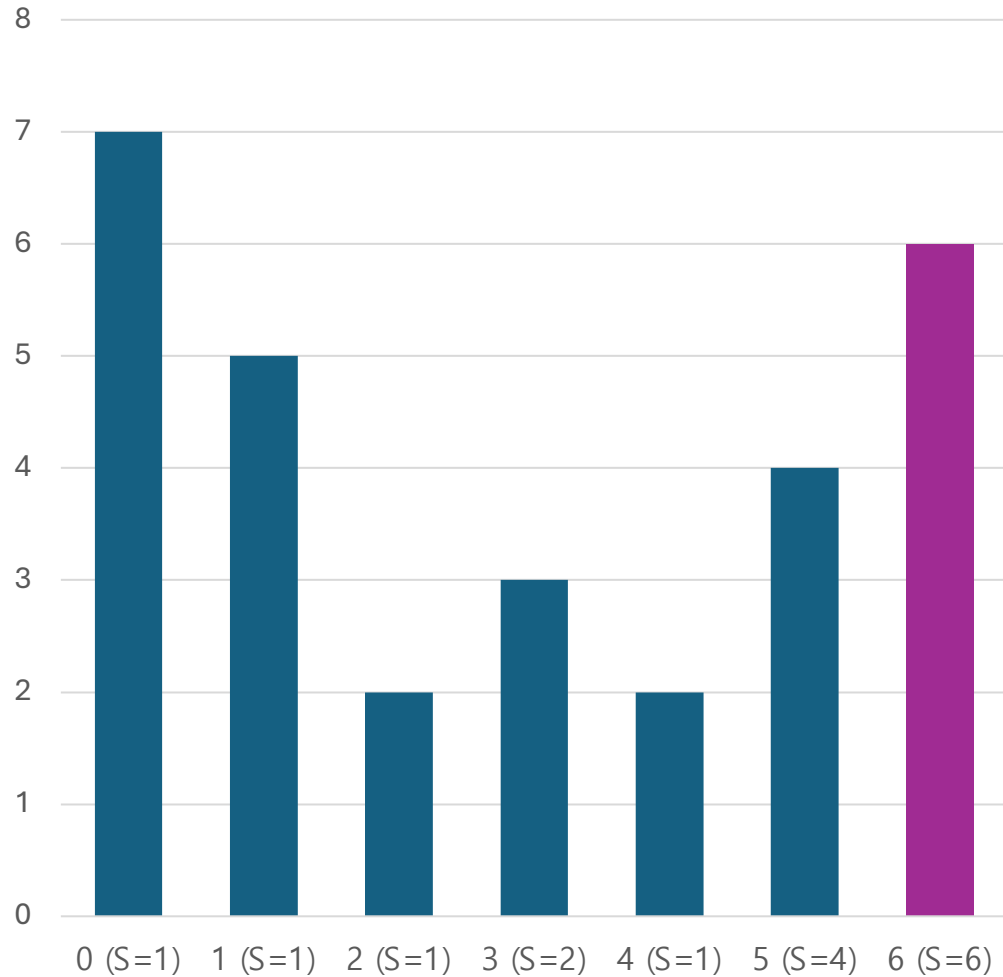
# Linear Algorithm



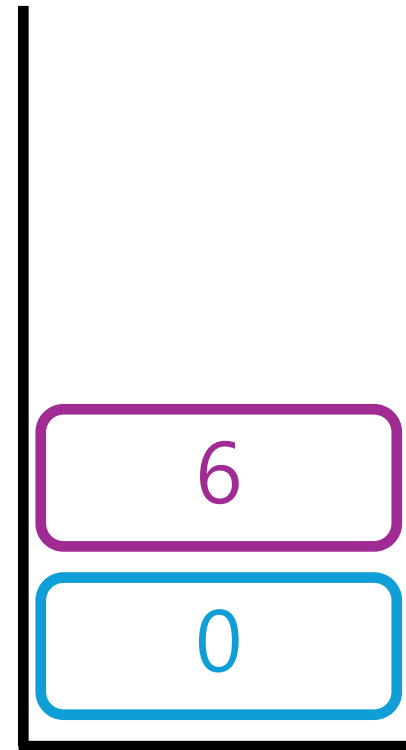
$$s_i := i - \text{stack.top}$$



# Linear Algorithm



$$s_i := i - \text{stack.top}$$



# Linear Algorithm Code



```
1  int *linear_span(int *price, int len) {
2      int *span = new int[len];
3      ArrayStack2<int> stack;
4
5      for (int i = 0; i < len; i++) {
6          while (!stack.empty() && price[stack.top()] <= price[i]) {
7              stack.pop();
8          }
9
10         if (stack.empty()) {
11             span[i] = i + 1;
12         }
13         else {
14             span[i] = i - stack.top();
15         }
16
17         stack.push(i);
18     }
19
20     return span;
21 }
```

# Linear Algorithm 분석

- 단위연산 : 현재 날짜의 가격과 stack의 마지막 원소에 해당하는 날짜의 **가격 비교**
- 입력크기 : 날짜의 길이 **len**
- 날짜별로 span을 계산하는 바깥 for문은 **len번 실행**
- stack에 들어가는 원소는 날짜별로 한번씩 들어가 **stack의 최대 크기는 len을 초과하지 않음**
- 단위연산의 횟수는 len을 초과하지 않고, **stack을 pop하기 위한 while문은 바깥 for문과 독립적으로 최악의 경우 len번 실행**
- **$W(n) = n + n \in O(n)$**