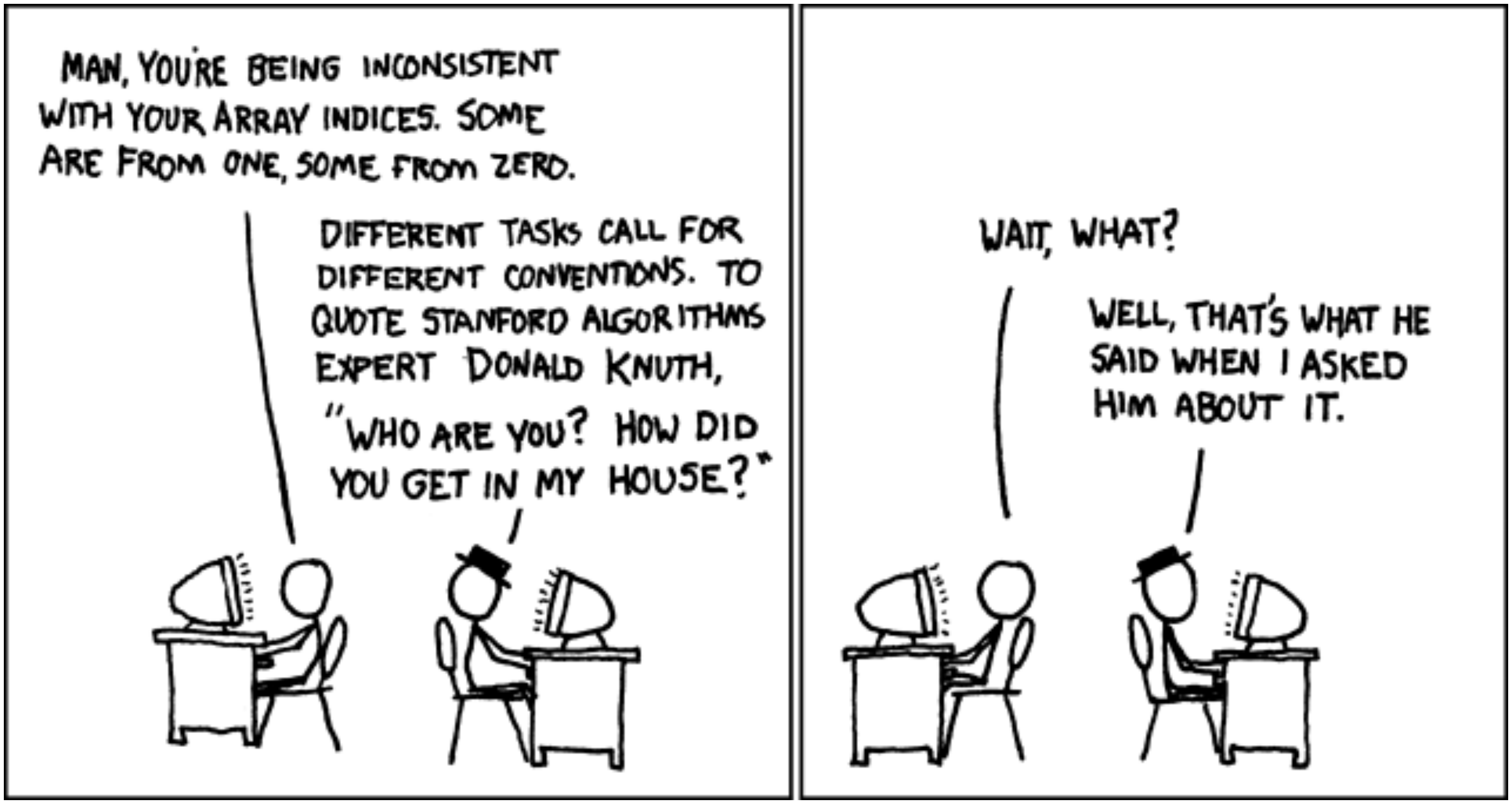
CSCI 104

Rafael Ferreira da Silva

rafsilva@isi.edu

Slides adapted from: Mark Redekopp and David Kempe

XKCD #163



Courtesy of Randall Munroe @ http://xkcd.com

LIST ADT & ARRAY-BASED IMPLEMENTATIONS

Lists

- Ordered collection of items, which may contain duplicate values, usually accessed based on their position (index)
  - Ordered = Each item has an index and there is a front and back (start and end)
  - Duplicates allowed (i.e. in a list of integers, the value 0 could appear multiple times)
  - Accessed based on their position (list[0], list[1], etc.)
- What are some operations you perform on a list?

List Operations

Operation	Description	Input(s)	Output(s)
insert	Add a new value at a particular location shifting others back	Index : int Value	
remove	Remove value at the given location	Index : int	Value at location
get / at	Get value at given location	Index : int	Value at location
set	Changes the value at a given location	Index : int Value	
empty	Returns true if there are no values in the list		bool
size	Returns the number of values in the list		int
push_back / append	Add a new value to the end of the list	Value	
find	Return the location of a given value	Value	Int : Index

IMPLEMENTATIONS

Implementation Strategies

- Linked List
  - Can grow with user needs
- Bounded Dynamic Array
  - Let user choose initial size but is then fixed
- Unbounded Dynamic Array
  - Can grow with user needs

BOUNDED DYNAMIC ARRAY STRATEGY

A Bounded Dynamic Array Strategy

- Allocate an array of some user-provided size
- What data members do I need?
- Together, think through the implications of each operation when using a bounded array (what issues could the fact that it is bounded cause)?

```
#ifndef BALISTINT_H
#define BALISTINT_H

class BAListInt {
 public:
  BAListInt(unsigned int cap);

  bool empty() const;
  unsigned int  size() const;
  void insert(int pos,
              const int& val);
  void remove(int pos);
  int const & get(int loc) const;
  int& get(int loc);
  void set(int loc, const int& val);
  void push_back(const int& val);
 private:



};
#endif
```

balist.h

A Bounded Dynamic Array Strategy

- What data members do I need?
  - Pointer to Array
  - Current size
  - Capacity
- Together, think through the implications of each operation when using a static (bounded) array
  - Push_back: Run out of room?
  - Insert: Run out of room, invalid location

```
#ifndef BALISTINT_H
#define BALISTINT_H

class BAListInt {
 public:
  BAListInt(unsigned int cap);

  bool empty() const;
  unsigned int  size() const;
  void insert(int pos,
              const int& val);
  void remove(int pos);
  int const & get(int loc) const;
  int& get(int loc);
  void set(int loc, const int& val);
  void push_back(const int& val);
 private:
  int* data_;
  unsigned int size_;
  unsigned int cap_;

};
#endif
```

balist.h

Implementation

- Implement the following member functions
  - A picture to help write the code

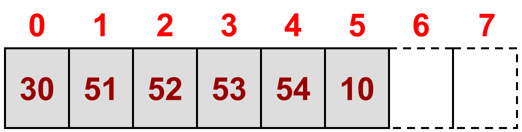
0	1	2	3	4	5	6	7
30	51	52	53	54	10		

```
BAListInt::BAListInt (unsigned int cap)
{


}

void BAListInt::push_back(const int& val)
{







}

void BAListInt::insert(int loc, const int& val)
{







}
```

balist.h (cont)

Implementation (cont.)

- Implement the following member functions
  - A picture to help write the code




```
void BAListInt::remove(int loc)
{



}
```

balist.h

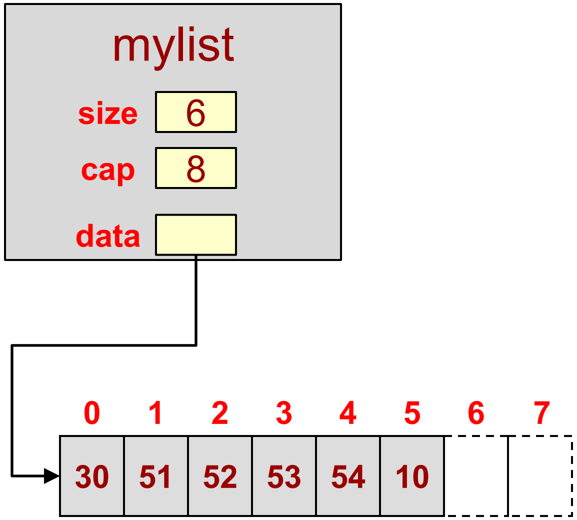
Constness

- What functions stand out as looking strange?
- Two versions of get()
- Why do we need two versions of get?
- Because we have two use cases…
  - 1. Just read a value in the array w/o changes
  - 2. Get a value w/ intention of changing it

```
#ifndef BALISTINT_H
#define BALISTINT_H

class BAListInt {
 public:
  BAListInt(unsigned int cap);

  bool empty() const;
  unsigned int  size() const;
  void insert(int pos,
              const int& val);
  bool remove(int pos);

  int& const  get(int loc) const;
  int& get(int loc);

  void set(int loc, const int& val);
  void push_back(const int& val);
 private:



};
#endif
```

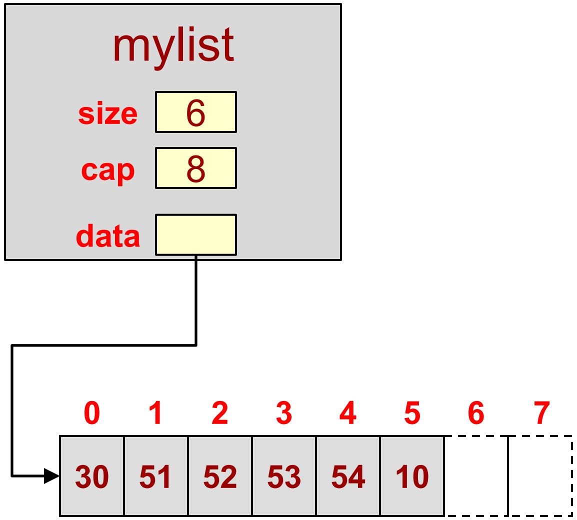
Constness

```
// ---- Recall List Member functions ------
// const version
int& const  BAListInt::get(int loc) const
{ return data_[i]; }

// non-const version
int& BAListInt::get(int loc)
{ return data_[i]; }

void BAListInt::insert(int pos, const int& val);


// ---- Now consider this code ------
void f1(const BAListInt& mylist)
{
  // This calls the const version of get
  // w/o the const-version this would not compile
  //  since mylist was passed as a const parameter
  cout << mylist.get(0) << endl;
  mylist.insert(0, 57); // won't compile..insert is non-const

}

int main()
{
  BAListInt mylist;
  f1(mylist);

}
```



Returning References

```
// ---- Recall List Member functions ------
// const version
int& const  BAListInt::get(int loc) const
{ return data_[i]; }

// non-const version
int& BAListInt::get(int loc)
{ return data_[i]; }

void BAListInt::insert(int pos, const int& val);

// ---- Now consider this code ------
void f1(BAListInt& mylist)
{
  // This calls the non-const version of get
  // if you only had the const-version this would not compile
  //  since we are trying to modify what the
  //  return value is referencing
  mylist.get(0) += 1;  // equiv. mylist.set(mylist.get(0)+1);
  mylist.insert(0, 57);
  // will compile since mylist is non-const
}
int main()
{ BAListInt mylist;
  f1(mylist);
}
```

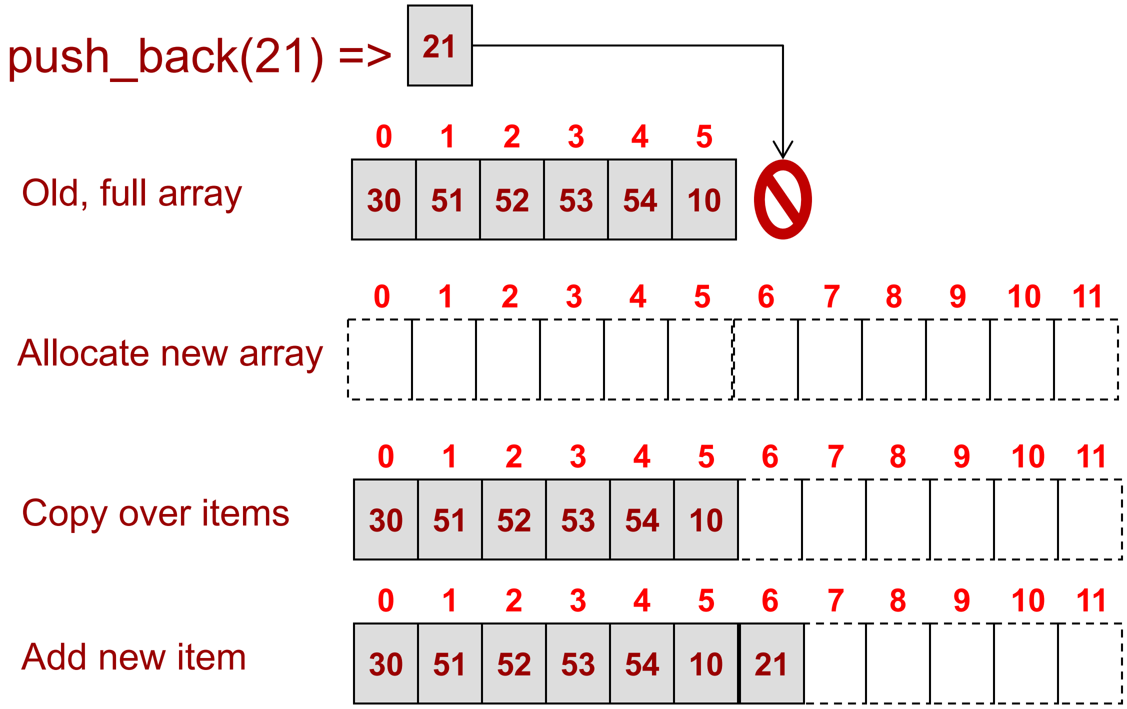


- **Moral of the Story: We need both versions of get()**

UNBOUNDED DYNAMIC ARRAY STRATEGY

Unbounded Array

- Any bounded array solution runs the risk of running out of room when we insert() or push_back()
- We can create an unbounded array solution where we allocate a whole new, larger array when we try to add a new item to a full array



We can use the strategy of allocating a new array ***twice*** the size of the old array

Activity

- What function implementations need to change if any?

```
#ifndef ALISTINT_H
#define ALISTINT_H

class AListInt  {
 public:
  bool empty() const;
  unsigned int  size() const;
  void insert(int loc,
              const int& val);
  void remove(int loc);
  int& const   get(int loc) const;
  int& get(int loc);
  void set(int loc, const int& val);
  void push_back(const T& new_val);
 private:


  int* _data;
  unsigned int _size;
  unsigned int _capacity;
};

// implementations here
#endif
```

Activity

- What function implementations need to change if any?

```
#ifndef ALISTINT_H
#define ALISTINT_H

class AListInt  {
 public:
  bool empty() const;
  unsigned int  size() const;
  void insert(int loc,
              const int& val);
  void remove(int loc);
  int& const   get(int loc) const;
  int& get(int loc);
  void set(int loc, const int& val);
  void push_back(const T& new_val);
 private:
  void resize(); // increases array size
  int* _data;
  unsigned int _size;
  unsigned int _capacity;
};

// implementations here
#endif
```

An Unbounded Dynamic Array Strategy

- Implement the push_back method for an unbounded dynamic array

```
#include "alistint.h"

void AListInt::push_back(const int& val)
{





}
```

alistint.cpp

```
void BAListInt::push_back(const int& val)
{
    if (size_ < cap_) {
        data_[size_++] = val;
    }
}
```

Previous code (Bounded)

An Unbounded Dynamic Array Strategy

- Implement the push_back method for an unbounded dynamic array

```
#include "alistint.h"

void AListInt::push_back(const int& val)
{
   if (_size >= _cap) {
      resize();
   }
   _data[_size++] = val;
}
```

alistint.cpp

```
void BAListInt::push_back(const int& val)
{
    if (size_ < cap_) {
        data_[size_++] = val;
    }
}
```

Previous code (Bounded)

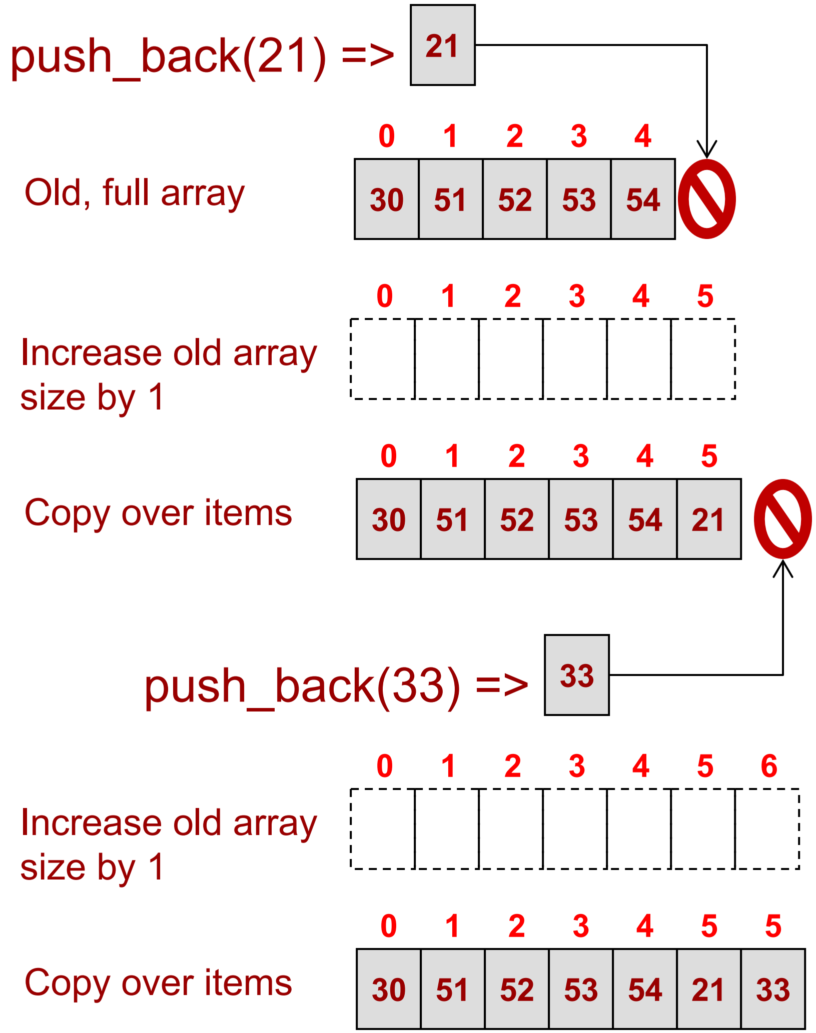
AMORTIZED RUNTIME

Example

- You love going to Universal Studios. You purchase a gold annual pass for $299. You visit Universal Studios once a month for a year. Each time you go you spend $20 on food, etc.
  - What is the cost of a visit?
- Your annual pass cost is spread or "**amortized**" (or averaged) over the duration of its usefulness
- Often times an operation on a data structure will have similar "irregular" costs that we can then amortize over future calls

Amortized Array Resize Run-time

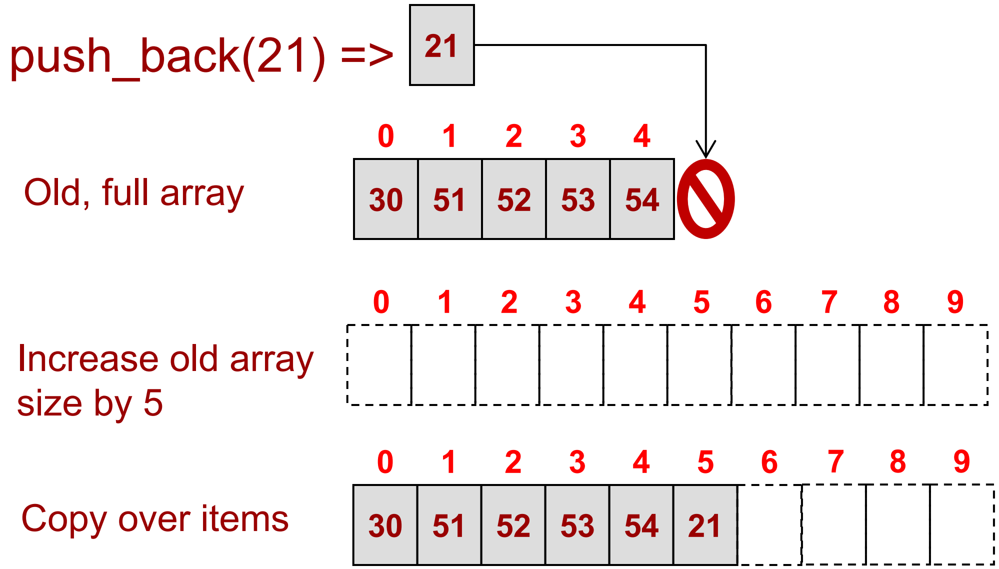
- What is the run-time of insert or push_back:
  - If we have to resize?
  - O(n)
  - If we don't have to resize?
  - O(1)
- Now compute the total cost of a series of insertions using resize by 1 at a time
- Each insert now costs O(n)… not good



Resize by 1 strategy

Amortized Array Resize Run-time

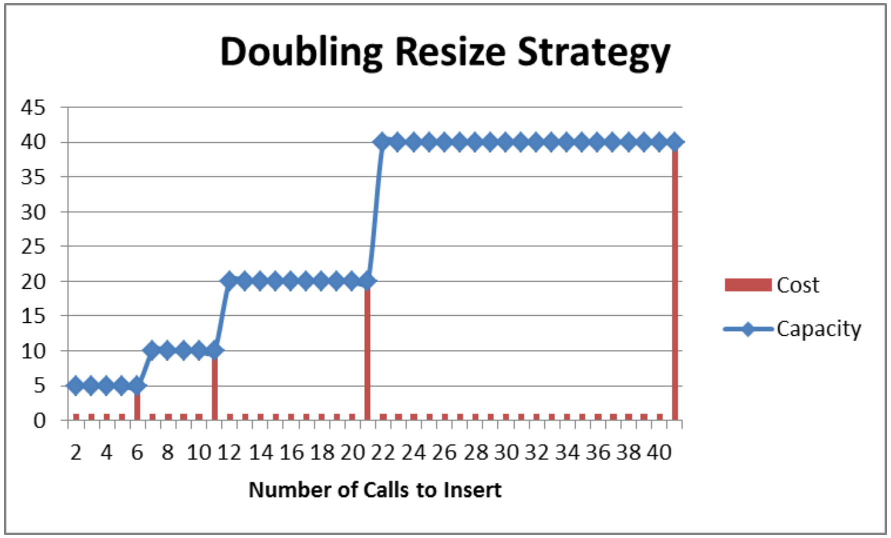
- What if we resize by adding 5 new locations each time
- Start analyzing when the list is full…
  - 1 call to insert will cost: 5
  - What can I guarantee about the next 4 calls to insert?
    - They will cost 1 each because I have room
  - After those 4 calls the next insert will cost: 10
  - Then 4 more at cost=1
- If the list is size n and full
  - Next insert cost = n
  - 4 inserts after than = 1 each
  - Cost for 5 inserts = n+5
  - Runtime = cost / insert = (n+5)/5 = O(n)



Resize by 5 strategy

Consider a Doubling Size Strategy

- Start when the list is full and at size n
- Next insertion will cost?
  - O(n+1)
- How many future insertions will be guaranteed to be cost = 1?
  - n-1 insertions
  - At a cost of 1 each, I get n-1 total cost
- So for the n insertions my total cost was
  - n+1 + n-1 = 2*n
- Amortized runtime is then:
  - Cost / insertions
  - O(2*n / n) = O(2)
    = O(1) = constant!!!



Another Example

- Let's say you are writing an algorithm to take a n-bit binary combination (3-bit and 4-bit combinations are to the right) and produce the next binary combination
- Assume all the cost in the algorithm is spent changing a bit (define that as 1 unit of work)
- I could give you any combination, what is the worst case run-time? Best-case?
  - O(n) => 011 to 100
  - O(1) => 000 to 001

3-bit Binary
000
001
010
011
100
101
110
111

4-bit Binary
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

Another Example

- Let's say you are writing an algorithm to take a n-bit binary combination (3-bit and 4-bit combinations are to the right) and produce the next binary combination
- Assume all the cost in the algorithm is spent changing a bit (define that as 1 unit of work)
- I could give you any combination, what is the worst case run-time? Best-case?
  - O(n) => 011 to 100
  - O(1) => 000 to 001

3-bit Binary	Cost
000	-
001	1
010	2
011	1
100	3
101	1
110	2
111	1

Worst Case: O(n log n)

n + floor(n/2) + floor(n/4) + ...

≤ n + n/2 + n/4 + ...

≤ 2n = O(n)