DS-210: PROGRAMMING FOR DATA SCIENCE

LECTURE 36

1. BINARY SEARCH

2. GREEDY ALGORITHMS

3. DIVIDE AND CONQUER

1. BINARY SEARCH

2. GREEDY ALGORITHMS

3. DIVIDE AND CONQUER

BINARY SEARCH

Input: sorted array/vector `v`

Task: find if a specific element `x` appears in it

BINARY SEARCH

Input: sorted array/vector `v`

Task: find if a specific element `x` appears in it

General strategy:

- maintain range `left..right` within `v` of where `x` could be
- repeat:
  - ask about the middle of the range
  - if `x` found, celebrate
  - if the middle element $<$ `x`, narrow search to the right half
  - otherwise, narrow search to the left half

LET'S IMPLEMENT IT!

LET'S IMPLEMENT IT!

Might be harder than you think!

Jon Bentley in "Programming pearls" claims that only 10% programmers succeed, even given unlimited amount of time

LET'S IMPLEMENT IT!

Might be harder than you think!

Jon Bentley in "Programming pearls" claims that only 10% programmers succeed, even given unlimited amount of time

Ad:

- Great book if you are interested in programming and basic algorithms
- I only know the first edition (mostly uses pseudocode)
- 2nd edition focused on C++ (probably still great)
- very cheap on eBay!

LET'S IMPLEMENT IT!

In [2]:
```
fn present(mut data:&[i32], element:i32) -> bool {
    let (mut left, mut right) = (0, data.len());
    // invariant: if element has not been found, it must be in data[left..right]
    while left < right {
        let middle = (left+right)/2;
        if data[middle] == element {
            return true
        } else if data[middle] > element {
            right = middle
        } else {
            left = middle + 1
        }
    }
    false
}
```

LET'S IMPLEMENT IT!

In [2]:
```
fn present(mut data:&[i32], element:i32) -> bool {
    let (mut left, mut right) = (0, data.len());
    // invariant: if element has not been found, it must be in data[left..right]
    while left < right {
        let middle = (left+right)/2;
        if data[middle] == element {
            return true
        } else if data[middle] > element {
            right = middle
        } else {
            left = middle + 1
        }
    }
    false
}
```

In [3]:
```
let v = vec![-16,-4,0,2,4,12,32,48,48,111];
```

In [4]:
```
present(&v, -16)
```

Out[4]: true

In [ ]:
```
present(&v, 32)
```

In [ ]:
```
present(&v, 5)
```

In [ ]:
```
present(&v, 1000)
```

LET'S IMPLEMENT IT!

In [2]:
```
fn present(mut data:&[i32], element:i32) -> bool {
    let (mut left, mut right) = (0, data.len());
    // invariant: if element has not been found, it must be in data[left..right]
    while left < right {
        let middle = (left+right)/2;
        if data[middle] == element {
            return true
        } else if data[middle] > element {
            right = middle
        } else {
            left = middle + 1
        }
    }
    false
}
```

In [3]:
```
let v = vec![-16,-4,0,2,4,12,32,48,48,111];
```

In [4]:
```
present(&v, -16)
```

Out[4]: true

In [5]:
```
present(&v, 5)
```

Out[5]: false

In [ ]:
```
present(&v, 1000)
```

In [ ]:
```
present(&v, 32)
```

LET'S IMPLEMENT IT!

In [2]:
```
fn present(mut data:&[i32], element:i32) -> bool {
    let (mut left, mut right) = (0, data.len());
    // invariant: if element has not been found, it must be in data[left..right]
    while left < right {
        let middle = (left+right)/2;
        if data[middle] == element {
            return true
        } else if data[middle] > element {
            right = middle
        } else {
            left = middle + 1
        }
    }
    false
}
```

In [3]:
```
let v = vec![-16,-4,0,2,4,12,32,48,48,111];
```

In [4]:
```
present(&v, -16)
```

Out[4]: true

In [5]:
```
present(&v, 5)
```

Out[5]: false

In [6]:
```
present(&v, 1000)
```

Out[6]: false

In [ ]:
```
present(&v, 32)
```

LET'S IMPLEMENT IT!

In [2]:
```
fn present(mut data:&[i32], element:i32) -> bool {
    let (mut left, mut right) = (0, data.len());
    // invariant: if element has not been found, it must be in data[left..right]
    while left < right {
        let middle = (left+right)/2;
        if data[middle] == element {
            return true
        } else if data[middle] > element {
            right = middle
        } else {
            left = middle + 1
        }
    }
    false
}
```

In [3]:
```
let v = vec![-16,-4,0,2,4,12,32,48,48,111];
```

In [4]:
```
present(&v, -16)
```

Out[4]: true

In [5]:
```
present(&v, 5)
```

Out[5]: false

In [6]:
```
present(&v, 1000)
```

Out[6]: false

In [7]:
```
present(&v, 32)
```

Out[7]: true

1. BINARY SEARCH

2. GREEDY ALGORITHMS

3. DIVIDE AND CONQUER

GREEDY ALGORITHMS

- Make locally best (or any) decision towards solving a problem

EXAMPLES WE SAW

- Heuristics for creating decision trees
    - select "best" single split and recurse

- Shortest paths (Dijkstra's algorithm)
    - select the vertex known to be closest
    - try routing paths through it
    - **Gives globally optimal solution!!!**

ADDITIONAL EXAMPLES

- Minimum spanning tree: find cheapest overall subset of edges so that the graph is connected
    - Kruskal's algorithm: keep adding the cheapest edge that connects disconnected groups vertices

ADDITIONAL EXAMPLES

- Minimum spanning tree: find cheapest overall subset of edges so that the graph is connected
    - Kruskal's algorithm: keep adding the cheapest edge that connects disconnected groups vertices

- Matching: matching conference attendees to available desserts they like
    - Another formulation: maximum size set of independent edges in graph
    - Keep adding edges as long as you can
    - This will give factor 2 approximation

1. BINARY SEARCH IN A SORTED VECTOR

2. GREEDY ALGORITHMS

3. DIVIDE AND CONQUER

DIVIDE AND CONQUER

If your problem is too difficult:

- partition it into subproblems
- solve the subproblems
- combine their solutions into a solution to the entire problem

DIVIDE AND CONQUER

If your problem is too difficult:

- partition it into subproblems
- solve the subproblems
- combine their solutions into a solution to the entire problem

Our plan: see two classic divide and conquer sorting algorithms

How would you do this?

MERGE SORT

Recursively:

- sort the first half
- sort the second half
- merge the results

MERGE SORT

Recursively:

- sort the first half
- sort the second half
- merge the results

Complexity for n elements?

MERGE SORT

Recursively:

- sort the first half
- sort the second half
- merge the results

Complexity for n elements?

- Merging two lists of Q and R elements takes $O(R + Q)$ time
- $O(\log n)$ levels of recursion: $O(n)$ work on each level
- $O(n \log n)$ time overall

IMPLEMENTING MERGING

In [8]:
```
fn merge(v1:Vec<i32>, v2:Vec<i32>) -> Vec<i32> {
    let (l1,l2) = (v1.len(), v2.len());
    let mut merged = Vec::with_capacity(l1+l2);
    let (mut i1, mut i2) = (0,0);
    while i1 < l1 {
        if (i2 == l2) || (v1[i1] <= v2[i2]) {
            merged.push(v1[i1]);
            i1 += 1;
        } else {
            merged.push(v2[i2]);
            i2 += 1;
        }
    }
    while i2 < l2 {
        merged.push(v2[i2]);
        i2 += 1;
    }
    merged
}
```

IMPLEMENTING MERGING

In [8]:
```rust
fn merge(v1:Vec<i32>, v2:Vec<i32>) -> Vec<i32> {
    let (l1,l2) = (v1.len(), v2.len());
    let mut merged = Vec::with_capacity(l1+l2);
    let (mut i1, mut i2) = (0,0);
    while i1 < l1 {
        if (i2 == l2) || (v1[i1] <= v2[i2]) {
            merged.push(v1[i1]);
            i1 += 1;
        } else {
            merged.push(v2[i2]);
            i2 += 1;
        }
    }
    while i2 < l2 {
        merged.push(v2[i2]);
        i2 += 1;
    }
    merged
}
```

In [9]:
```rust
let v1 = vec![3,4,8,11,12];
let v2 = vec![1,2,3,9,22];
merge(v1,v2)
```

Out[9]:
```
[1, 2, 3, 3, 4, 8, 9, 11, 12, 22]
```

IMPLEMENTING MERGE SORT

In [10]:
```
fn merge_sort(input:&[i32]) -> Vec<i32> {
    if input.len() <= 1 {
        input.to_vec()
    } else {
        let split = input.len() / 2;
        let v1 = merge_sort(&input[..split]);
        let v2 = merge_sort(&input[split..]);
        merge(v1,v2)
    }
}
```

In [11]:
```
let v = vec![2,4,21,6,2,32,62,0,-2,8];
merge_sort(&v)
```

Out[11]: [-2, 0, 2, 2, 4, 6, 8, 21, 32, 62]

QUICK SORT

- Select an arbitrary (random?) element x
- Partition your vector:
    - Move elements lower than x to the left
    - Move elements greater than x to the right
- Sort the left part and right part

QUICK SORT

- Select an arbitrary (random?) element x
- Partition your vector:
    - Move elements lower than x to the left
    - Move elements greater than x to the right
- Sort the left part and right part

Complexity for n elements?

QUICK SORT

- Select an arbitrary (random?) element x
- Partition your vector:
    - Move elements lower than x to the left
    - Move elements greater than x to the right
- Sort the left part and right part

Complexity for n elements?

- Partitioning k elements takes $O(k)$ time
- Intuition: The size of the problem usually decreases by constant factor in a recursive call
- Expected time: $O(n \log n)$ time overall

IMPLEMENTING PARTITIONING

In [12]:

```
fn partition(input:&mut [i32], pivot: i32) -> (usize,usize) {
    // move numbers lower than pivot to the left
    let mut left = 0;
    for i in 0..input.len() {
        if input[i] < pivot {
            input.swap(i,left);
            left += 1;
        }
    }
    // input[..left]: numbers lower than pivot

    // move numbers greater than pivot to the right
    let mut right = input.len();
    for i in (left..input.len()).rev() {
        if input[i] > pivot {
            right -= 1;
            input.swap(i,right);
        }
    }
    // input[right..]: numbers greater than pivot

    (left,right)
}
```

IMPLEMENTING QUICKSORT

In [13]:
```
:dep rand
use rand::Rng;

fn quicksort(input:&mut [i32]) {
    if input.len() >= 2 {
        // pivot = random element from the input
        let pivot = input[rand::thread_rng().gen_range(0..input.len())];

        let (left,right) = partition(input,pivot);

        quicksort(&mut input[..left]);
        quicksort(&mut input[right..]);
    }
}
```

In [14]:
```
let mut v = vec![145,12,3,7,83,12,8,64];
quicksort(&mut v);
v
```

Out[14]:
```
[3, 7, 8, 12, 12, 64, 83, 145]
```