CS2040S
Data Structures and Algorithms

Welcome!

ARCHIPELAGO

is open

How to Search!

Algorithm Analysis

- Big-O Notation
- Model of computation

Searching

Peak Finding

- 1-dimension
- 2-dimensions

Admin: Tutorial/Recitations

Recitation Allocation Appeals:

- Almost done
- 4 students left

Please respond to emails requesting your schedule.

Tutorial Allocation Appeals:

- In progress
- Hopefully done by tomorrow.

Both start next week (Week 3)

Puzzle (2?) of the Week (Contest)

Treasure Island

You have found a treasure chest! It has a lot of locks on it!

You need ALL the correct keys to open the chest.

Find the right set of keys to win!

Image Credit: Britannica

Puzzle of the Week (Contest)

Treasure Island

You choose a set of keys.

You can test whether all the good keys are in the set (and possibly some extra keys). If even one bad key is in the set, then it fails.

Goal: identify exactly the good keys.

Image Credit: Britannica

Puzzle of the Week (Contest)

Treasure Island

You choose a set of keys.

You can test whether all the good keys are in the set (and possibly some extra keys). If even one bad key is in the set, then it fails.

Goal: identify exactly the good keys.

Puzzle of the Week (Contest)

Treasure Island

You choose a set of keys.

You can test whether all the good keys are in the set (and possibly some extra keys). If even one bad key is in the set, then it fails.

Goal: identify exactly the good keys.

FAIL

Puzzle of the Week (Contest)

Treasure Island

You choose a set of keys.

You can test whether all the good keys are in the set (and possibly some extra keys). If even one bad key is in the set, then it fails.

Goal: identify exactly the good keys.

SUCCEED

Puzzle of the Week (Contest)

Treasure Island

You choose a set of keys.

You can test whether all the good keys are in the set (and possibly some extra keys). If even one bad key is in the set, then it fails.

Goal: identify exactly the good keys.

SUCCEED and OPEN

Puzzle of the Week (Contest)

Treasure Island

Two goals:

- Minimize number of tries.
- Minimize total number of keys tried.

Image Credit: Britannica

Puzzle of the Week (Contest)

Treasure Island

For example:

- Total number of keys = 1024
- Number of correct keys = 17

Required number of attempts: >= 122

Image Credit: Britannica

Announcements

Competition:

Find the keys!

Open on Coursemology for one week:

- Optional.
- Write a program to implement your best treasure finding strategy.
- We will test it on various treasure chests.
- Fewest tries / fewest keys wins!
- (And a small bonus for participating.)

How to Search!

Algorithm Analysis

- Big-O Notation
- Model of computation

Searching

Peak Finding

- 1-dimension
- 2-dimensions

Last time…

Binary Search

- Simple, ubiquitous algorithm.
- Surprisingly easy to add bugs.
- Some ideas for avoiding bugs:
  - Problem specification
  - Preconditions
  - Postconditions
  - Invariants / loop invariants
  - Validate (when feasible)

Sorted array: `A[0..n-1]`

2	4	4	5	6	7	8	9	11	17	23	28

```
int search(A, key, n)
    begin = 0
    end = n-1
    while begin < end do:
        mid = begin + (end-begin)/2;
        if  key <= A[mid] then
            end = mid
        else begin = mid+1
    return (A[begin]==key) ? begin : -1
```

Binary Search

Sorted array: A[0..n-1]

2	4	4	5	6	7	8	9	11	17	23	28

Not just for searching arrays:

- Assume a complicated function:

  int complicatedFunction(int s)

- Assume the function is always increasing:

  complicatedFunction(i) < complicatedFunction(i+1)

- Find the minimum value j such that:

  complicatedFunction(j) > 100

A problem…

Tutorial allocation

A problem...

Tutorial allocation

Tutorials
(in order
of tutor
preference)

T1

T2

T3

T4

T5

T6

T7

T8

A problem...

Tutorial allocation

Tutorials
(in order
of tutor
preference)

Students want
certain tutorials.

A problem...

Tutorial allocation

Tutorials
(in order
of tutor
preference)

Students want
certain tutorials.

We want each
tutorial to have
< 18 students..

A problem...

Tutorial allocation

Tutorials
(in order of tutor preference)

Students want certain tutorials.

We want each tutorial to have < 18 students..

How many tutorials do we need to run?

A problem...

Tutorial allocation

Tutorials
(in order of tutor preference)

Students want certain tutorials.

We want each tutorial to have < 18 students..

How many tutorials do we need to run?

A problem...

Tutorial allocation

Tutorials
(in order of tutor preference)

Can we do greedy allocation?

First, fill T1.
Then fill T2.
Then fill T3.
…
Stop when all students are allocated

A problem...

Tutorial allocation

Can we do greedy allocation?

NO

Assume max tutorial size is 1.

Assign student 1 to tutorial 1.

Now we need all 8 tutorials.

A problem...

Tutorial allocation

Can we do greedy allocation?

NO

Assume max tutorial size is 1.

Assign student 1 to tutorial 1.

Now one student has no feasible allocation!

A problem...

Example of decomposing an algorithm into parts!

Tutorial allocation

Assume we can solve allocation problem:

Given a fixed set of tutorials and a fixed set of students, find an allocation where every student has a slot.

Assumptions:

- may be > 18 students in a slot!
- minimizes max students in a slot.

A problem…

Tutorial allocation

Tutorials
(in order of tutor preference)

How to find minimum number of tutorials that we need to open to ensure: no tutorial has more than 18 students.

A problem...

Tutorial allocation

Tutorials
(in order of tutor preference)

Observation:

Number of students in BIGGEST tutorial only decreases as number of tutorials increases.

Monotonic function of number of tutorials!

A problem...

Tutorial allocation

Tutorials
(in order
of tutor
preference)

Monotonic function of number of tutorials!

⇩

Binary Search

A problem...

Tutorial allocation

Solution:

Binary Search

Define:

MaxStudents(x) = number of students in most crowded tutorial, if we offer x tutorials.

Binary Search

MaxStudents(x) = number of students in most crowded tutorial, if we offer x tutorials.

```
Search(n)
    begin = 0
    end = n-1
    while begin < end do:
        mid = begin + (end-begin)/2;
        if  MaxStudents(mid) <= 18 then
            end = mid
        else begin = mid+1
    return begin
```

Binary Search

Sorted array: A[0..n-1]

2	4	4	5	6	7	8	9	11	17	23	28

Not just for searching arrays:

- Assume a complicated function:

  int complicatedFunction(int s)

- Assume the function is always increasing:

  complicatedFunction(i) < complicatedFunction(i+1)

- Find the minimum value j such that:

  complicatedFunction(j) > 100

How to Search!

Algorithm Analysis

- Big-O Notation
- Model of computation

Searching

Peak Finding

- 1-dimension
- 2-dimensions

Peak Finding

Input: Some function f(x)

Peak Finding

Global Maximum for Optimization problems:

- Find a good solution to a problem.
- Find a design that uses less energy.
- Find a way to make more money.
- Find a good scenic viewpoint.
- Etc.

Why local maximum?

- Finds a *good enough* solution.
- Local maxima are close to the global maximum?
- Much, much faster.

Global Maximum

Input: Array A[0..n-1]

Output: global maximum element in A

How long to find a global maximum?

Input: Array A[0..n-1]

Output: maximum element in A

1. $O(\log n)$
2. $O(n)$
3. $O(n \log n)$
4. $O(n^2)$
5. $O(2^n)$

ARCHIPELAGO
is open

Global Maximum

Unsorted array: `A[0..n-1]`

7	4	9	2	11	6	23	4	28	8	17	5

```
FindMax(A,n)
    max = A[1]
    for i = 1 to n do:
        if (A[i]>max) then max=A[i]
```

Too slow!

Time Complexity: $O(n)$

Peak (Local Maximum) Finding

Input: Some function f(x)

Output: A local maximum

Peak Finding

Input: Some ~~function~~ array A[0..n-1]

7	4	9	2	11	6	23	4	6	8	17	21

Output: a local maximum in A

$$A[i-1] \leq A[i] \textbf{ and } A[i+1] \leq A[i]$$

Assume that

$$A[-1] = A[n] = -MAX_INT$$

Peak Finding: Algorithm 1

Input: Some array `A[0..n-1]`

2	4	9	2	11	6	23	4	6	8	17	5

FindPeak

- Start from A[1]
- Examine every element
- Stop when you find a peak.

Peak Finding: Algorithm 1

Input: Some array `A[0..n-1]`

2	4	9	2	11	6	23	4	6	8	17	5

FindPeak

- Start from A[1]
- Examine every element
- Stop when you find a peak.

Peak Finding: Algorithm 1

Input: Some array `A[0..n-1]`

2	4	9	2	11	6	23	4	6	8	17	5

FindPeak

- Start from A[1]
- Examine every element
- Stop when you find a peak.

Peak Finding: Algorithm 1

Input: Some array `A[0..n-1]`

2	4	9	2	11	6	23	4	6	8	17	5

Running time: n

Simple improvement?

Peak Finding: Algorithm 1

Input: Some array `A[0..n-1]`

2	2	3	4	5	6	9	11	13	15	17	25

Start in the middle!

Worst-case: n/2

Peak Finding: Algorithm 2

Reduce-and-Conquer

Start in the middle

5 < 6? ← **OK**

6 > 23? ← **NO**

Recurse!

Peak Finding: Algorithm 2

Reduce-and-Conquer

Peak Finding: Algorithm 2

Divide-and-Conquer

Peak Finding: Algorithm 2

Divide-and-Conquer

Peak Finding

Input: Some array A[0..n-1]

2	4	9	2	11	6	23	4	6	8	17	5

FindPeak(A, n)

if A[n/2] is a peak **then return** n/2

else if A[n/2+1] > A[n/2] **then**

Search for peak in right half.

else if A[n/2–1] > A[n/2] **then**

Search for peak in left half.

Peak Finding

Input: Some array A[0..n-1]

2	4	9	2	11	6	23	4	6	8	17	5

FindPeak(A, n)

if A[n/2] is a peak **then return** n/2

else if A[n/2+1] > A[n/2] **then**

FindPeak (A[n/2+1..n], n/2)

else if A[n/2–1] > A[n/2] **then**

FindPeak (A[1..n/2-1], n/2)

Peak Finding

Is this correct?

2	4	9	2	11	6	23	4	6	8	17	5

FindPeak(A, n)

if A[n/2] is a peak **then return** n/2

else if A[n/2+1] > A[n/2] **then**

FindPeak (A[n/2+1..n], n/2)

else if A[n/2–1] > A[n/2] **then**

FindPeak (A[1..n/2-1], n/2)

Should this be >=?

Missing else condition?

Peak Finding

Should this be >=? No: recurse on the larger half.

2	4	9	2	11	6	23	4	6	8	17	5

FindPeak(A, n)

if A[n/2] is a peak **then return** n/2

else if A[n/2+1] > A[n/2] **then**

FindPeak (A[n/2+1..n], n/2)

else if A[n/2–1] > A[n/2] **then**

FindPeak (A[1..n/2-1], n/2)

Peak Finding

Missing else condition? No: else we have found a peak!

2	4	9	2	11	6	23	4	6	8	17	5

FindPeak(A, n)

if A[n/2] is a peak **then return** n/2

else if A[n/2+1] > A[n/2] **then**

FindPeak (A[n/2+1..n], n/2)

else if A[n/2–1] > A[n/2] **then**

FindPeak (A[1..n/2-1], n/2)

Peak Finding

Missing else condition? No: else we have found a peak!

2	4	9	2	11	6	23	4	6	8	17	5

FindPeak(A, n)

if A[n/2+1] > A[n/2] **then**

FindPeak (A[n/2+1..n], n/2)

else if A[n/2–1] > A[n/2] **then**

FindPeak (A[1..n/2-1], n/2)

else A[n/2] is a peak; **return** n/2

Peak Finding

Key property ➔ invariant:

If we recurse in the right half, then there exists a peak in the right half.

2	4	9	2	5	6	23	4	6	8	17	5

Peak Finding

Key property:

- If we recurse in the right half, then there exists a peak in the right half.

Explanation:

- Assume there is "no peak" in the right half.
- Given: A[middle] < A[middle + 1]
- Since no peaks, A[middle+1] < A[middle+2]
- Since no peaks, A[middle+2] < A[middle+3]
- …
- Since no peaks, A[n-1] < A[n] ← PEAK!!

Peak Finding

Recurse on right half, since 23 > 6.

Assume no peaks in right half.

Peak Finding

Key property:

- If we recurse in the right half, then there exists a peak in the right half.

Explanation:

- Assume there is "no peak" in the right half.
- Because we recursed right: A[middle] < A[middle + 1]
- Since no peaks, A[middle+1] < A[middle+2]
- Since no peaks, A[middle+2] < A[middle+3]
- …
- Since no peaks, A[n-1] < A[n] ← PEAK!!

Peak Finding

Key property:

- If we recurse in the right half, then there exists a peak in the right half.

Induction:

- Assume there is "no peak" in the right half.
- Inductive hypothesis:

  For all (j > middle): A[j-1] < j

Peak Finding

Key property:

- If we recurse in the right half, then there exists a peak in the right half.

Induction:

- Assume there is "no peak" in the right half.
- Inductive hypothesis:

  For all (j > middle): A[j-1] < j

- Base case: j = middle+1

  Because we recursed on the right half, we know that A[middle] < A[middle + 1].

Peak Finding

Key property:

- If we recurse in the right half, then there exists a peak in the right half.

Induction:

- Assume there is "no peak" in the right half.
- Inductive hypothesis:

  For all (j > middle): A[j-1] < j
- Induction: j > middle+1

  By induction, A[j-2] <= A[j-1].

  If A[j-1] >= A[j], then A[j-1] is a peak ➔ contradiction.

Peak Finding

Key property:

- If we recurse in the right half, then there exists a peak in the right half.

Induction:

- Assume there is "no peak" in the right half.
- Inductive hypothesis:

  For all (j > middle): A[j-1] < j
- Conclusion: A[n-2] < A[n-1]

  ➔ A[n-1] is a peak ➔ contradiction.

Key Invariants:

Correctness:

There exists a peak in the range [begin, end].

Key Invariants:

Is this good enough to prove the algorithm works?

There exists a peak in the range [begin, end].

ARCHIPELAGO
is open

Key Invariants:

Not good enough to prove the algorithm works!

There exists a peak in the range [begin, end].

Key Invariants:

Not good enough to prove the algorithm works!

There exists a peak in the range [begin, end].

Key Invariants:

Not good enough to prove the algorithm works!

There exists a peak in the range [begin, end].

11	12	13	14	15

Run peak finding algorithm → returns 15

Key Invariants:

Not good enough to prove the algorithm works!

There exists a peak in the range [begin, end].

2	4	9	2	11	12	13	14	15	18	17	5

Run peak finding algorithm → returns 15

But 15 is NOT a peak!

If the recursive call finds a peak, is it still a peak after the recursive call returns?

Key Invariants:

Correctness:

1. There exists a peak in the range [begin, end].

2. Every peak in[begin, end] is a peak in [1, n].

Peak Finding

Key property:

- If we recurse in the right half, then every peak in the right half is a peak in the array.

Proof: use the invariant (inductively)

- Immediately true for every peak that is not at a boundary in new array.

Peak Finding

Key property:

- If we recurse in the right half, then every peak in the right half is a peak in the array.

Proof: use the invariant (inductively)

- True by invariant for current array.

Peak Finding

Key property:

- If we recurse in the right half, then every peak in the right half is a peak in the array.

Proof: use the invariant (inductively)

- If 45 is a peak in the new array but not the old array, then we would not recurse on the right side.
  ➔ If left edge is a peak in new array, then it is a peak.

2	4	9	2	11	12	60	45	23	30	17	5

If 45 is a peak in right half and we recurse on right half, then it is a peak.

Key Invariants:

Correctness:

1. There exists a peak in the range [begin, end].

2. Every peak in[begin, end] is a peak in [1, n].

Peak Finding

Running time?

2	4	9	2	11	6	23	4	6	8	17	5

FindPeak(A, n)

if A[n/2] is a peak **then return** n/2

else if A[n/2+1] > A[n/2] **then**

Search for peak in right half.

else if A[n/2–1] > A[n/2] **then**

Search for peak in left half.

Peak Finding

Running time:

Time for comparing A[n/2] with neighbors

Time to find a peak in an array of size n

Recursion

$$T(n) = T(n/2) + \theta(1)$$

Peak Finding

Running time:

Time for comparing A[n/2] with neighbors

Time to find a peak in an array of size n

Recursion

$$T(n) = T(n/2) + \theta(1)$$

Unrolling the recurrence:

$$T(n) = \theta(1) + \theta(1) + \ldots + \theta(1) = O(\log n)$$

Peak Finding

Unrolling the recurrence:

Rule:
T(X) = T(X/2) + O(1)

$$T(n) = T(n/2) + \theta(1)$$

$$= T(n/4) + \theta(1) + \theta(1)$$

$$= T(n/8) + \theta(1) + \theta(1) + \theta(1)$$

$$\ldots$$

$$\ldots$$

$$= T(1) + \theta(1) + \ldots + \theta(1) =$$

$$= \theta(1) + \theta(1) + \ldots + \theta(1) =$$

Peak Finding

Unrolling the recurrence:

$$T(n) = T(n/2) + \theta(1)$$

$$= T(n/4) + \theta(1) + \theta(1)$$

$$= T(n/8) + \theta(1) + \theta(1) + \theta(1)$$

$$\ldots$$

$$\ldots$$

$$= T(1) + \theta(1) + \ldots + \theta(1) =$$

$$= \theta(1) + \theta(1) + \ldots + \theta(1) =$$

Rule:
T(X) = T(X/2) + O(1)

Number of times you can divide n by 2 until you reach 1.

Peak Finding

How many times can you divide a number ***n*** in half before you reach 1?

$$\underbrace{2 \times 2 \times \ldots \times 2}_{\log(n)} = 2^{\log(n)} = n$$

Note: I always assume $\log = \log_2$

$$O(\log_2 n) = O(\log n)$$

Peak Finding

Running time:

Time for comparing A[n/2] with neighbors

Time to find a peak in an array of size n

Recursion

$$T(n) = T(n/2) + \theta(1)$$

Unrolling the recurrence:

$$T(n) = \underbrace{\theta(1) + \theta(1) + \ldots + \theta(1)}_{\log(n)} = O(\log n)$$

Peak Finding

Input: Some array A[0..n-1]

7	4	9	2	11	6	23	4	6	8	8	21

Output: a local maximum in A

$A[i-1] \leq A[i]$ **and** $A[i+1] \leq A[i]$

Assume that

$A[-1] = A[n] = -MAX_INT$

Steep Peaks

Input: Some array A[0..n-1]

7	4	9	2	11	6	23	4	6	8	8	5

Output: a local maximum in A

`A[i-1]<A[i]` **and** `A[i+1]<A[i]`

Assume that

`A[-1] = A[n] = -MAX_INT`

Steep Peaks

Input: Some array A[0..n-1]

7	4	9	2	11	6	23	4	6	8	8	5

Output: a local maximum in A

`A[i-1]<A[i]` **and** `A[i+1]<A[i]`

Can we find *steep* peaks efficiently (in O(log n) time) using the same approach?