CSE 417
Binary Search (pt 2)

UNIVERSITY *of* WASHINGTON

Reminders

- **HW1 is due Wednesday**
  - some clarifications on Piazza
    - assume cost formula is correct for any sizes, even zero
  - notice the structure of these graphs...

Outline for Today

- Generalized binary search

Example 1: who broke the code?

Most groups working on software have a shared repository that they all update the code in. Consider this situation:

```
9:04 AM  Alice submits new code
9:38 AM  Bob submits new code
9:45 AM  Charlie submits new code

... 1000 more submissions ...

6:35 PM  Alice submits new code
```

everything works

something is broken

Example 1: who broke the code?

The usual tools will let you make a copy of the repository at any point in the past, i.e., after any of submissions.

Q: How do we figure out which submission broke it?

A: Use binary search

- copy the repository with 500 new submissions… see if it works
- if it does, try 750 submissions
- if not, try 250 submissions
- repeat until we know the last submission that worked
  - the one after that broke it

Example 2: where is the bug?

Suppose that the following code computes the wrong answer:

```
int v = ...;
```

looks correct

```
// Invariant: P(i, v)
for (int i = 0; i < 1000; i++) {
  ...
}
```

```
return v;
```

wrong answer

Example 2: where is the bug?

At some i, we are computing the wrong value of v.

Q: How do we find the iteration that hits the bug?

A: Use binary search

- set a breakpoint to stop when i = 500… see if v is correct
- if it is, try 750
- if not, try 250
- repeat until we find the value of i that hits the bug

> As before, easy to get into the state where i = whatever

Generalized Binary Search

Let's see how to describe binary search in a way that is broad enough to cover this case as well…

Generalized Binary Search

Input: A monotonically increasing function $f : Z \rightarrow R$, a range [a, b), and a number x in R

- Z means the integers
- R can be any ordered set

Output: integer t in [a, b] such that:

- $f(s) <= x$ for all s in [a, t)
- $x < f(s)$ for all s in [t, b)

Generalized Binary Search

Example: sorted array

- f(s) = A[s]
- f is monotonically increasing because A is sorted

Example: who broke the code

- f maps a submission number to {0, 1}
- 0 if the code works, 1 if the code is broken
- f is monotonically increasing because all submissions after the bad one leave it in a broken state

Exercise: where is the bug

Implementing Generalized Binary Search

```
int i = a, j = b;

// Invariant: f(a), ..., f(i-1) <= x and x < f(j), ..., f(b-1)
while (i < j) {
  int m = (i + j) / 2;
  if (f(m) <= x)
    i = m + 1;
  else
    j = m;
}

return i;
```

only these two lines of code changed

Aside: Java 8 Lambdas

We can represent a general function of this type in Java 8 with

```
java.util.function.IntFunction<R>
```

This represents a function from integers to type R

- call `apply(int)` to invoke the function
- can pass in a function using lambdas, e.g., "`x -> 2*x + 1`"

Example: Java 8 Lambdas

We could define

```
public int binarySearch(IntFunction<Double> f, int a, int b, double x);
```

and then call

```
binarySearch(x -> Math.tanh(x), 0, 1000, 0.9);
```

or even

```
binarySearch(Math::tanh, 0, 1000, 0.9);
```

Generalized Binary Search Run Time

- If each call to f takes T(n) time (for some n), then generalized binary search takes O(T(n) log(b – a)) time

- The log(b – a) factor is usually very small
  - often so small as to be negligible
  - some theoretical analysis even ignores such factors…

Generalized Binary Search

If $f : Z \rightarrow R$ is monotonically increasing, binary search lets us take an x in R and find the t such that $f(t) = x$ (if one exists) — i.e., *inverting* f

Theorem: If $f : Z \rightarrow R$ is a monotonically increasing function on $[a, b]$ that we can compute in $T(n)$ time, then we can compute f^{-1} in time $O(T(n) \log(b - a))$

- if $b - a = O(n^k)$, then $\log(b - a) = k \log n = O(\log n)$
- the $\log(b - a)$ factor is often so small as to be negligible, so we can compute f^{-1} in essentially the same time when f is monotonic

This hints at why binary search is so widely useful…

Foreword

Inverting functions is usually **difficult**. In particular, NP-complete problems are inverses of functions that are efficiently computable.

In terms of what can be computed *efficiently*, inverting f is...

impossible	if f^{-1} is NP-complete
free	if f is monotonic

(Many possibilities in between these two.)

Example 3: breaking even

HW1 Problem 1: given costs A, B_S, H, and sizes M_S,
find the cheapest way to manufacture all of the jean sizes

- model as a shortest path problem

Q: Find the maximum hemming cost H at which we still break even

- assume we have projected sales for our jeans, so we can project revenue
- question: how small does H need to be for manufacturing costs <= revenue
- (leave A, B_S, and M_S fixed)

A: binary search

Example 3: breaking even

Define f(H) = cheapest way to manufacture designer jeans
with costs A, B_S, H and sizes M_S

> Some complicated function…
  - at H = 0, cost for using the cheapest size for every smaller size
  - at H = infinity, cost for buying every size separately
  - can’t describe it with a formula BUT we **can compute it**

> It is monotonically increasing
  - cost of each way of computing increases as H increases
  - minimum of those numbers can only increase as well

Example 3: breaking even

Define f(H) = cheapest way to manufacture designer jeans
with costs A, B_S, H and sizes M_S

- Can compute monotonically increasing f => can compute f^{-1}
  - binary search range [0, T], where T is large enough that no hemming is done
    - use "**repeated doubling**" to find T in log T calls to f as well
  - total cost is O(log T) times cost to compute manufacturing cost (f)

Example 3: breaking even

To further see how binary search can come up in surprising places, imagine starting with the question about break-even H

- I.e., without having just seen how to compute the the manufacturing costs

> In general, if something looks hard to compute, see if we can write it as the inverse of some monotonic function...

- in this case, see that we can compute the (cheapest) manufacturing costs
- then see that these depend monotonically on H
- can get more creative in other examples...