

DSC 40B

Theoretical Foundations II

Searching a Database

Databases

- Large data sets are often stored in **databases**.

PID	FullName	Level
A1843	Wan Xuegang	SR
A8293	Deveron Greer	SR
A9821	Vinod Seth	FR
A8172	Aleix Bilbao	JR
A2882	Kayden Sutton	SO
A1829	Raghu Mahanta	FR
A9772	Cui Zemin	SR
⋮	⋮	⋮

Query

- ▶ What is the name of the student with PID A8172?

Linear Search

- ▶ We *could* answer this with a linear search.
- ▶ Recall worst-case time complexity: $\Theta(n)$.
- ▶ Is there a better way?

Theoretical Lower Bounds

- ▶ **Given:** an array `arr` and a target `t`, determine the index of `t` in the array.
- ▶ Lower bound: $\Omega(n)$
 - ▶ `linear_search` has the best possible worst-case complexity!

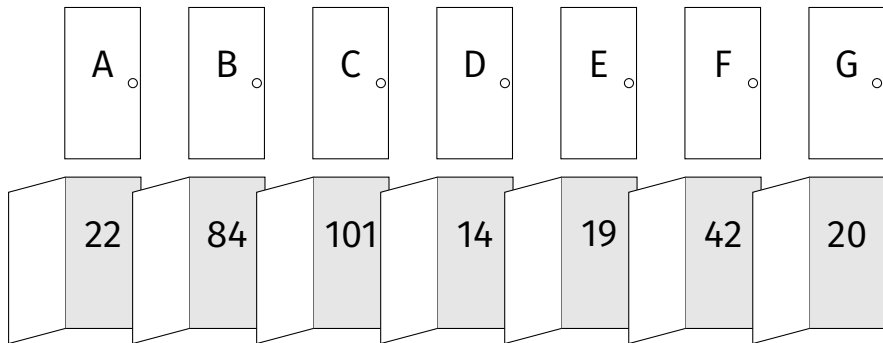
Theoretical Lower Bounds

- ▶ **Given:** an **sorted** array `arr` and a target `t`, determine the index of `t` in the array.
- ▶ This is an **easier** problem.
- ▶ Lower bound: $\Omega(?)$

DSC 40B

Theoretical Foundations II

Binary Search



Game Show

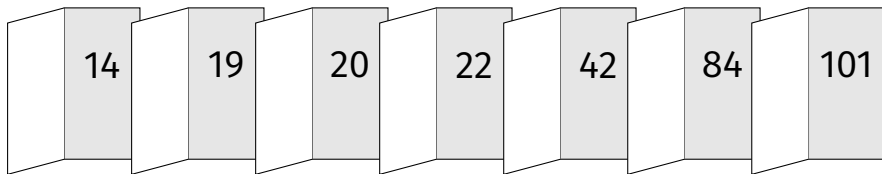
- ▶ **Goal:** guess the door with number 42 behind it.
- ▶ **Caution:** with every wrong guess, your money is reduced by half.

Strategy

- ▶ Can't do much better than linear search.
 - ▶ "Is it door A?"
 - ▶ "OK, is it door B?"
 - ▶ "Door C?"
- ▶ After an incorrect first guess, the right door could be any of the other $n - 1$ doors!

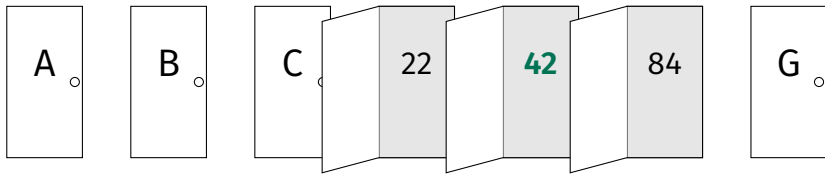
But now...

- Suppose the host tells you that the numbers are **sorted** in increasing order.



Exercise

Which door do you pick first?



Strategy

- ▶ First pick the middle door.
- ▶ Allows you to rule out half of the other doors.
- ▶ Pick door in the middle of what remains.
- ▶ Repeat, **recursively**.

In Code

- ▶ Given a **sorted** array `arr` and target `t`, find the index of `t` in `arr`.

`t = 21`

-10	-6	-3	1	2	5	12	21	33	35	42
0	1	2	3	4	5	6	7	8	9	10

Exercise

Suppose you know t is in `arr[a:b]`. How do we calculate the next index to guess?

-10	-6	-3	1	2	5	12	21	33	35	42
0	1	2	3	4	5	6	7	8	9	10

Binary Search

```
import math
def binary_search(arr, t, start, stop):
    """
    Searches arr[start:stop] for t.
    Assumes arr is sorted.
    """
    if stop - start <= 0:
        return None
    middle = math.floor((start + stop)/2)
    if arr[middle] == t:
        return middle
    elif arr[middle] > t:
        return binary_search(arr, t, start, middle)
    else:
        return binary_search(arr, t, middle+1, stop)
```

```

import math
def binary_search(arr, t, start, stop):
    """
    Searches arr[start:stop] for t.
    Assumes arr is sorted.
    """
    if stop - start <= 0:
        return None
    middle = math.floor((start + stop)/2)
    if arr[middle] == t:
        return middle
    elif arr[middle] > t:
        return binary_search(arr, t, start, middle)
    else:
        return binary_search(arr, t, middle+1, stop)

```

t = 21

-10	-6	-3	1	2	5	12	21	33	35	42
0	1	2	3	4	5	6	7	8	9	10

DSC 40B

Theoretical Foundations II

Thinking Inductively

Recursion

- ▶ Recursive algorithms can almost look like **magic**.
- ▶ How can we be sure that `binary_search` works?

Tips

1. Make sure algorithm works in the **base case**.
2. Check that all recursive calls are on **smaller problems**.
3. **Assuming** that the recursive calls work, does the whole algorithm work?

Base Case

- ▶ Smallest input for which you can easily see that the algorithm works.
- ▶ Recursion works by making problem smaller until base case is reached.
- ▶ Usually $n = 0$ or $n = 1$ (or even both!)

Exercise

What is the base case for `binary_search`?

Base Case: $n = 0$

- ▶ Suppose `arr[start:stop]` is empty.
- ▶ In this case, the function returns **None**.
 - ▶ **Correct!**

Base Case: $n = 1$

- ▶ Suppose `arr[start:stop]` has one element.
- ▶ If that element is the target, the algorithm will find it.
 - ▶ **Correct!**
- ▶ If it isn't, the algorithm will recurse on a problem of size 0 and return **None**.
 - ▶ **Correct!**

Recursive Calls

- ▶ Recursive calls must be on **smaller problems**.
 - ▶ Otherwise, base case never reached. Infinite recursion!

```
import math
def binary_search(arr, t, start, stop):
    """
    Searches arr[start:stop] for t.
    Assumes arr is sorted.
    """
    if stop - start <= 0:
        return None
    middle = math.floor((start + stop)/2)
    if arr[middle] == t:
        return middle
    elif arr[middle] > t:
        return binary_search(arr, t, start, middle)
    else:
        return binary_search(arr, t, middle+1, stop)
```

- ▶ Is arr[start:middle] smaller than arr[start:stop]?
- ▶ Is arr[middle+1:stop] smaller than arr[start:stop]?

Leap of Faith

- ▶ **Assume** the recursive calls work.
- ▶ Does the overall algorithm work, then?

```
import math
def binary_search(arr, t, start, stop):
    """
    Searches arr[start:stop] for t.
    Assumes arr is sorted.
    """
    if stop - start <= 0:
        return None
    middle = math.floor((start + stop)/2)
    if arr[middle] == t:
        return middle
    elif arr[middle] > t:
        return binary_search(arr, t, start, middle)
    else:
        return binary_search(arr, t, middle+1, stop)
```

Induction

- ▶ These steps can be turned into a formal proof by **induction**.
- ▶ For us, less necessary to prove to other people.
- ▶ Instead, prove to **yourself** that your code works.
- ▶ We won't be doing formal inductive proofs.

Why does this work?

- ▶ Show that it works for size 1 (base case).
- ▶ \implies will work for size 2 (inductive step).
- ▶ \implies will work for sizes 3, 4 (inductive step).
- ▶ \implies will work for sizes 5, 6, 7, 8 (inductive step).

DSC 40B

Theoretical Foundations II

Recurrence Relations

Time Complexity of Binary Search

- ▶ What is the time complexity of `binary_search`?
- ▶ No loops!

Best Case

```
import math
def binary_search(arr, t, start, stop):
    """
    Searches arr[start:stop] for t.
    Assumes arr is sorted.
    """
    if stop - start <= 0:
        return None
    middle = math.floor((start + stop)/2)
    if arr[middle] == t:
        return middle
    elif arr[middle] > t:
        return binary_search(arr, t, start, middle)
    else:
        return binary_search(arr, t, middle+1, stop)
```

Worst Case

Let $T(n)$ be worst case time on input of size n .

```
import math
def binary_search(arr, t, start, stop):
    """
    Searches arr[start:stop] for t.
    Assumes arr is sorted.
    """
    if stop - start <= 0:
        return None
    middle = math.floor((start + stop)/2)
    if arr[middle] == t:
        return middle
    elif arr[middle] > t:
        return binary_search(arr, t, start, middle)
    else:
        return binary_search(arr, t, middle+1, stop)
```

Recurrence Relations

- We found

$$T(n) = \begin{cases} T(n/2) + \Theta(1), & n \geq 2 \\ \Theta(1), & n = 1 \end{cases}$$

- This is a **recurrence relation**.

Solving Recurrences

- ▶ We want simple, non-recursive formula for $T(n)$ so we can see how fast $T(n)$ grows.
 - ▶ Is it $\Theta(n)$? $\Theta(n^2)$? Something else?
- ▶ Obtaining a simple formula is called **solving** the recurrence.

Example: Getting Rich

- ▶ Suppose on day 1 of job, you are paid \$3.
- ▶ Each day thereafter, your pay is doubled.
- ▶ Let $S(n)$ be your pay on day n :

$$S(n) = \begin{cases} 2 \cdot S(n-1), & n \geq 2 \\ 3, & n = 1 \end{cases}$$

Example: Unrolling

$$S(n) = \begin{cases} 2 \cdot S(n - 1), & n \geq 2 \\ 3, & n = 1 \end{cases}$$

- Take $n = 4$.

Solving Recurrences

We'll use a four-step process to solve recurrences:

1. “Unroll” several times to find a pattern.
2. Write general formula for k th unroll.
3. Solve for # of unrolls needed to reach base case.
4. Plug this number into general formula.

Step 1: Unroll several times

$$S(n) = \begin{cases} 2 \cdot S(n - 1), & n \geq 2 \\ 3, & n = 1 \end{cases}$$

Step 2: Find general formula

$$\begin{aligned} S(n) &= 2 \cdot S(n - 1) \\ &= 2 \cdot 2 \cdot S(n - 2) \\ &= 2 \cdot 2 \cdot 2 \cdot S(n - 3) \end{aligned}$$

On step k :

Step 3: Find step # of base case

- ▶ On step k , $S(n) = 2^k \cdot S(n - k)$.
- ▶ When do we see $S(1)$?

Step 4: Plug into general formula

- ▶ From step 2: $S(n) = 2^k \cdot S(n - k)$.
- ▶ From step 3: Base case of $S(1)$ reached when $k = n - 1$.
- ▶ So:

Solving the Recurrence

- ▶ We have **solved** the recurrence:

$$S(n) = 3 \cdot 2^{n-1}$$

- ▶ This is the **exact** solution. The **asymptotic** solution is $S(n) = \Theta(2^n)$.
- ▶ We'll call this method “solving by unrolling”.

DSC 40B

Theoretical Foundations II

Binary Search Recurrence

Binary Search

- ▶ What is the time complexity of `binary_search`?
- ▶ Best case: $\Theta(1)$.
- ▶ Worst case:

$$T(n) = \begin{cases} T(n/2) + \Theta(1), & n \geq 2 \\ \Theta(1), & n = 1 \end{cases}$$

Simplification

- ▶ When solving, we can replace $\Theta(f(n))$ with $f(n)$:

$$T(n) = \begin{cases} T(n/2) + 1, & n \geq 2 \\ 1, & n = 1 \end{cases}$$

- ▶ As long as we state final answer using Θ notation!

Another Simplification

- ▶ When solving, we can assume n is a power of 2.

Step 1: Unroll several times

$$T(n) = \begin{cases} T(n/2) + 1, & n \geq 2 \\ 1, & n = 1 \end{cases}$$

Step 2: Find general formula

$$\begin{aligned}T(n) &= T(n/2) + 1 \\&= T(n/4) + 2 \\&= T(n/8) + 3\end{aligned}$$

On step k :

Step 3: Find step # of base case

- ▶ On step k , $T(n) = T(n/2^k) + k$
- ▶ When do we see $T(1)$?

Step 4: Plug into general formula

- ▶ $T(n) = T(n/2^k) + k$
- ▶ Base case of $T(1)$ reached when $k = \log_2 n$.
- ▶ So:

Note

- ▶ Remember: $\log_b x = (\log_a x) / (\log_a b)$
- ▶ So we don't write $\Theta(\log_2 n)$
- ▶ Instead, just: $\Theta(\log n)$

Time Complexity of Binary Search

- ▶ Best case: $\Theta(1)$
- ▶ Worst case: $\Theta(\log n)$

Is binary search fast?

- ▶ Suppose all 10^{19} grains of sand are assigned a unique number, sorted from least to greatest.
- ▶ Goal: find a particular grain.
- ▶ Assume one basic operation takes 1 nanosecond.
- ▶ Linear search: 317 years.
- ▶ Binary search: ≈ 60 nanoseconds.

Exercise

Binary search seems so much faster than linear search. What's the caveat?

Caveat

- ▶ The array must be **sorted**.
- ▶ This takes $\Omega(n)$ time.

Why use binary search?

- ▶ If data is not sorted, sorting + binary search takes longer than linear search.
- ▶ But if doing multiple queries, looking for nearby elements, sort once and use binary search after.

Theoretical Lower Bounds

- ▶ A lower bound for searching a sorted list is $\Omega(\log n)$.
- ▶ This means that binary search has optimal worst case time complexity.

Databases

- ▶ Some database servers will **sort** by key, use binary search for queries.
- ▶ Though most commonly, **B-Trees** are used.