Search Algorithms and Order of Efficiency

CSET 3150

Common Problems

- There are some very common problems that we use computers to solve:
  - **Searching** through a lot of records for a specific record or set of records
  - **Sorting**, or placing records in a desired order
- At times we need to use both of these techniques as part of solving the same problem.

Common Problems

- There are numerous algorithms to perform searches and sorts.
- Over the remaining lessons in this course, we will briefly explore a few common search and sort algorithms.
  - We begin with search algorithms as applied to simple arrays.
  - Techniques can be extended to arrays of structures.

Search Algorithms

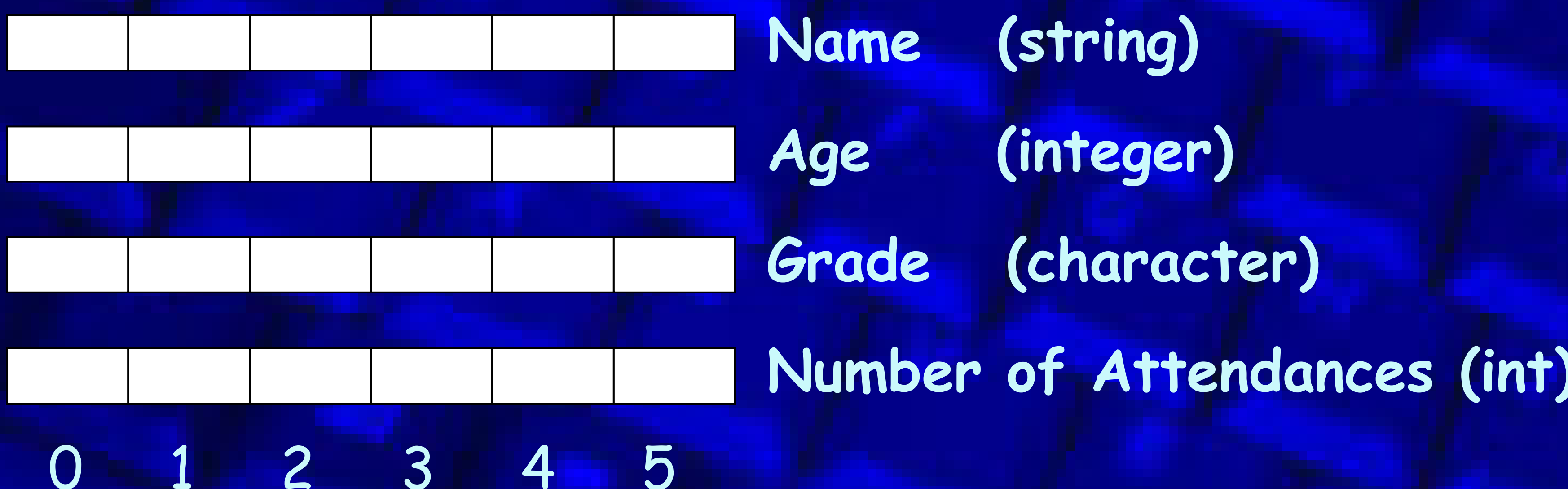
- Search: A search algorithm is a method of locating a specific item of information in a larger collection of data.
- There are two primary algorithms used for searching the contents of an array:
  - Linear or Sequential Search
  - Binary Search

Linear Search

- This is a very simple algorithm.
- It uses a loop to sequentially step through an array, starting with the first element.
- It compares each element with the value being searched for (key) and stops when that value is found or the end of the array is reached.

An Example: Parallel Arrays

Corresponding position in each array refers to a different piece of data which is an item of data belonging to the same logical entity, e.g.,



Searching an Array of structs

- Same process as for a simple array
- Use one of the fields to search

  e.g., birthdays[listindex].month

- Returns position in array as before

Linear Search

- Algorithm pseudocode:

```
set found to false; set position to –1; set index to 0
while index < number of elemts. and found is false
    if list[index] is equal to search value
        found = true
        position = index
    end if
    add 1 to index
end while
return position
```

Linear Search Function

- Linear searching is easy to program

```
int itemInArray(char item, char[] arr, int validEntries)
{
    int index = -1;

    for (int i = 0; i < validEntries; i++)
    {
        if (item == arr[i])
            index = i;
    }
    return index;
}
```

Linear Search Example

- Array `numlist` contains:

17	23	5	11	2	29	3

- Searching for the the value 11, linear search examines 17, 23, 5, and 11
- Searching for the the value 7, linear search examines 17, 23, 5, 11, 2, 29, and 3

Linear Search Tradeoffs

- Benefits:
  - Easy algorithm to understand
  - Array can be in any order
- Disadvantages:
  - Inefficient (slow): for array of N elements, examines N/2 elements on average for value in array, N elements for value not in array

Binary Search

Requires array elements to be in order

1. Divides the array into three sections:
   - middle element
   - elements on one side of the middle element
   - elements on the other side of the middle element
2. If the middle element is the correct value, done. Otherwise, go to step 1. using only the half of the array that may contain the correct value.
3. Continue steps 1. and 2. until either the value is found or there are no more elements to examine

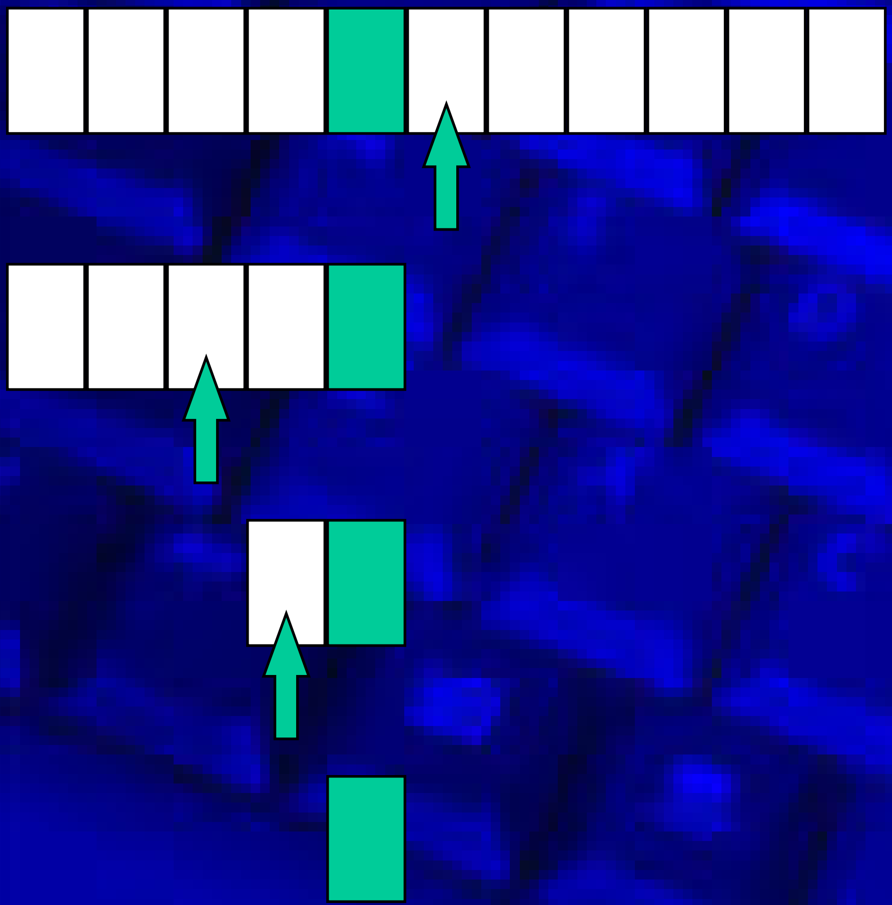
Binary Search Example

- Array `numlist2` contains:

2	3	5	11	17	23	29

- Searching for the the value 11, binary search examines 11 and stops
- Searching for the the value 7, linear search examines 11, 3, 5, and stops

How a Binary Search Works



Always look at the center value. Each time you get to discard half of the remaining list.

Is this fast ?

Binary Search Program

```
/* This program demonstrates the binarySearch function,
   that performs a binary search on an integer array. */

#include <stdio.h>

/* Function prototype */
int binarySearch(int [], int, int);

/* constant for array size */
const int arrSize = 20;
```

declarations

Binary Search Program

```
int main(void)
{
    int tests[] = {101, 142, 147, 189, 199, 207, 222, 234, 289, 296,
                   310, 319, 388, 394, 417, 429, 447, 521, 536, 600};
    int results, empID;
    printf("Enter the Employee ID you wish to search for: ");
    scanf("%d", &empID);
    results = binarySearch(tests, arrSize, empID);
    if (results == -1)
        printf("That number does not exist in the array.\n");
    else
    {
        printf("That ID is found at element %d", results);
        printf(" in the array\n");
    }
    return 0;
}
```

main function

Binary Search Program

```
int binarySearch(int array[], int numElems, int value)
{
    int first = 0, last = numElems - 1, middle, position = -1;
    int found = 0;
    while (!found && first <= last)
    {       middle = (first + last) / 2;                /* Calculate mid point */
            if (array[middle] == value)           /* If value is found at mid */
            {   found = 1;
                position = middle;
            }
            else if (array[middle] > value)     /* If value is in lower half */
                 last = middle - 1;
            else
                first = middle + 1;               /* If value is in upper half */
    }
    return position;
}
```

search function

How Fast is a Binary Search?

- Worst case: 11 items in the list took 4 tries
- How about the worst case for a list with 32 items ?
  - 1st try - list has 16 items
  - 2nd try - list has 8 items
  - 3rd try - list has 4 items
  - 4th try - list has 2 items
  - 5th try - list has 1 item

How Fast is a Binary Search?

List has 250 items

1st try - 125 items
2nd try - 63 items
3rd try - 32 items
4th try - 16 items
5th try - 8 items
6th try - 4 items
7th try - 2 items
8th try - 1 item

List has 512 items

1st try - 256 items
2nd try - 128 items
3rd try - 64 items
4th try - 32 items
5th try - 16 items
6th try - 8 items
7th try - 4 items
8th try - 2 items
9th try - 1 item

What's the Pattern?

- List of 11 took 4 tries
- List of 32 took 5 tries
- List of 250 took 8 tries
- List of 512 took 9 tries

- $32 = 2^5$ and $512 = 2^9$
- $8 < 11 < 16$ $\quad$ $2^3 < 11 < 2^4$
- $128 < 250 < 256$ $\quad$ $2^7 < 250 < 2^8$

A Very Fast Algorithm!

- How long (worst case) will it take to find an item in a list 30,000 items long?

$2^{10} = 1024$	$2^{13} = 8192$
$2^{11} = 2048$	$2^{14} = 16384$
$2^{12} = 4096$	$2^{15} = 32768$

- So, it will take only 15 tries!

$Log_2(n)$ Efficiency

- We say that the binary search algorithm runs in **$\log_2(n)$** time.
  - also written as **lg(n)**
- $Log_2(n)$ means the log to the base 2 of some value of n.
- $8 = 2^3$ $\log_2(8) = 3$ $16 = 2^4$ $\log_2(16) = 4$
- There are no algorithms that run faster than $\log_2(n)$ time.

Binary Search Tradeoffs

- Benefits:
  - Much more efficient than linear search. For array of n elements, performs at most $\log_2(n)$ comparisons
- Disadvantages:
  - Requires that array elements be sorted

Searching

- A question you should always ask when selecting a search algorithm is "How fast does the search have to be?"
  - The reason is that, in general, the faster the algorithm is, the more complex it is.
- Bottom line: you don't always need to use or should use the fastest algorithm.

Binary Search "Game"

- Number guessing game:
  - Program selects a number in a range
  - Player guesses
  - Program feedback "low" or "high"
  - Player guesses
  - Repeat until allowed number of guesses is reached or number is guessed

Comparing Algorithms

- Before we can compare different methods of searching (or sorting, or any algorithm), we need to think a bit about the time requirements for the algorithm to complete its task.
- We could also compare algorithms by the amount of memory needed
  - For the code
  - For execution (work space)

Comparing Algorithms

- An algorithm can require different times to solve different problems of the same size (a measure of efficiency)
- For example, the time it takes an algorithm to search for the integer '1' in an array of 100 integers depends on the nature of the array
  - are they sorted already?
  - if so, '1' may be at the start or end

Order: A Comparison Tool

- Most of the time we consider the maximum amount of time that an algorithm can require
- We call this worst-case analysis
- Worst-case analysis states that an algorithm is $O(f(n))$ if it will not take anymore time than $k * f(n)$ time units for all but a finite number of values n.
- Read the 'big-O', O(…), as 'on the order of'
- $f(n)$ is a function describing how the time or memory requirements increase with increasing problem size (increasing values of n).

Order

- The worst-case scenario doesn't mean the algorithm will always be slow, but that it is guaranteed never to take more time then the given bound
- This is called an asymptotic bound
  - Remember those asymptotes from algebra (same thing)
- Sometimes, the worst-case happens very rarely (if at all) in practice

Average Performance

- A harder to calculate metric is an algorithm's average-case performance
- Average-case analysis uses probabilities of problem sizes and problems of a given size to determine how it will act on average
- We won't worry about calculating the average-case performance at this point

Sequential Search

- If the item we are looking for is the first item, the search is O(1).
  - This is the best-case scenario
- If the target item is the last item (item n), the search takes O(n).
  - This is the worst-case scenario.
- On average, the item will tend to be near the middle (n/2) but this can be written ($\frac{1}{2}$*n), and as we will see, we can ignore multiplicative coefficients. Thus, the average-case is still O(n)

Sequential Search

- So, the time that sequential search takes is proportianal to the number of items to be searched
- Another way of saying the same thing using the Big-O notation is:
  - $O(n)$
  - A sequential search is of order n

Binary Search

- We also have looked at the binary search algorithm
- How much more efficient (if at all) is a binary search when compared to a sequential search?
- We can use "order" to help find the answer

Binary Search

- Considering the worst-case for binary search:
  - We don't find the item until we have divided the array as far as it will divide
- We first look at the middle of n items, then we look at the middle of $n/2$ items, then $n/2^2$ items, and so on…
- We will divide until $n/2^k = 1$, k is the number of times we have divided the set (when we have divided all we can, the above equation will be true)
- $n/2^k = 1$ when $n = 2^k$, so to find out how many times we divided the set, we solve for k

  $k = \log_2 n$
- Thus, the algorithm takes $O(\log n)$, the worst-case (we ingore logarithmic base)

Comparing Search Algorithms

- We know
  - sequential search is $O(n)$ worst-case
  - binary search is $O(\log_2 n)$ worst-case
- Which is better?
- Given $n = 1{,}000{,}000$ items
  - $O(n) = O(1{,}000{,}000)$ /* sequential */
  - $O(\log_2 n) = O(19)$ /* binary */
- Clearly binary search is better in worst-case for large values of n, but there is always trade-offs that must be considered
  - Binary search requires the array to be sorted
  - If the item to be found is near the extremes of the array, sequential may be faster