O Notation 2

Searching

Searching

- It is often useful to find out whether or not a list contains a particular item
  - Such a search can either return true or false
  - Or the position of the item in the list
- If the array isn't sorted use *linear search*
  - Start with the first item, and go through the array comparing each item to the target
  - If the target item is found return true (or the index of the target element)

Linear Search

```
int linearSearch(int arr[], int n, int x){
   for (int i=0; i < n; i++){
      if(arr[i] == x){
         return i;
      }
   } //for
   return -1; //target not found
}
```

The function returns as soon as the target item is found

return -1 to indicate that the item has not been found

Worst case cost function: $t_{\text{linear search}} = 3n+3$

Linear Search Barometer Instruction

- Search an array of *n* items
- The barometer instruction is equality checking (or *comparisons* for short)
  - `arr[i] == x;`
  - There are actually two other barometer instructions
    - What are they?
- How many comparisons does linear search perform?

the *barometer operation* is the most frequently executed operation

look for barometer operations in loops, or functions, or recursive calls

```
int linearSearch(int arr[], int n, int x){
     for (int i=0; i < n; i++){
          if(arr[i] == x){
               return i;
          }
     } //for
     return -1; //target not found
}
```

Linear Search Comparisons

- Best case
  - The target is the first element of the array
  - Makes 1 comparison
- Worst case
  - The target is not in the array or
  - The target is at the last position in the array
  - Makes *n* comparisons in either case
- Average case
  - Is it (*best case* + *worst case*) / 2, i.e. (*n* + 1) / 2?

Linear Search: Average Case

- There are two situations when the worst case occurs
  - When the target is the last item in the array
  - When the target is not there at all
- To calculate the average cost we need to know how often these two situations arise
  - We can make assumptions about this
  - Though these assumptions may not hold for a particular use of linear search

Assumptions

- A1: The target is not in the array half the time
  - Therefore half the time the entire array has to be checked to determine this
- A2: There is an equal probability of the target being at any array location
  - If it is in the array
  - That is, there is a probability of 1/*n* that the target is at some location *i*

Cost When Target Not Found

- Work done if the target is *not* in the array
  - *n* comparisons
  - This occurs with probability of 0.5 (A1)

Cost When Target Is Found

- Work done if target is in the array:
  - 1 comparison if target is at the 1st location
    - Occurs with probability 1/*n* (A2)
  - 2 comparisons if target is at the 2nd location
    - Also occurs with probability 1/*n*
  - *i* comparisons if target is at the *i*th location
- Take the weighted average of the values to find the total expected number of comparisons (*E*)
  - $E = 1*1/n + 2*1/n + 3*1/n + \ldots + n * 1/n$ *or*
  - $E = (n + 1) / 2$

Average Case Cost

- Target is *not* in the array: n comparisons
- Target *is* in the array $(n + 1) / 2$ comparisons
- Take a weighted average of the two amounts:
  - $= (n * ½) + ((n + 1) / 2 * ½)$
  - $= (n / 2) + ((n + 1) / 4)$
  - $= (2n / 4) + ((n + 1) / 4)$
  - $= (3n + 1) / 4$
- Therefore, on average, we expect linear search to perform $(3n + 1) / 4$ comparisons

Linear Search and Sorted Arrays

- If we sort the target array first we can change the linear search average cost to approximately *n* / 2
  - Once a value equal to or greater than the target is found the search can end
    - So, if a sequence contains 8 items, on average, linear search compares 4 of them,
    - If a sequence contains 1,000,000 items, linear search compares 500,000 of them, etc.
- However, if the array is sorted, it is possible to do *much better* than this by using binary search

Binary Search Algorithm

```
int binarySearch(int arr[], int n, int x){
    int low = 0;
    int high = n - 1;
    int mid = 0;
    while (low <= high){
        mid = (low + high) / 2;
        if(x == arr[mid]){
            return mid;
        } else if(x > arr[mid]){
            low = mid + 1;
        } else { //x < arr[mid]
            high = mid - 1;
        }
    } //while
    return -1; //target not found
}
```

Index of the last element in the array

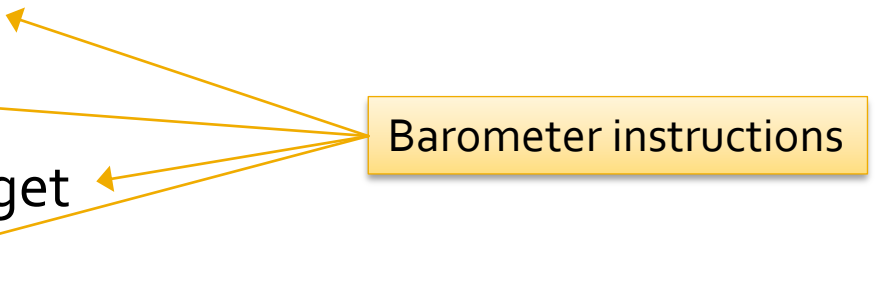
Note: if, else if, else

Analyzing Binary Search

- The algorithm consists of three parts
  - Initialization (setting lower and upper)
  - While loop including a return statement on success
  - Return statement which executes on failure
- Initialization and return on failure require the same amount of work regardless of input size
- The number of times that the while loop iterates depends on the size of the input

Binary Search Iteration

- The while loop contains an *if*, *else if*, *else* statement
- The first if condition is met when the target is found
  - And is therefore performed at most once each time the algorithm is run
- The algorithm usually performs 5 operations for each iteration of the while loop
  - Checking the while condition
  - Assignment to mid
  - Equality comparison with target
  - Inequality comparison
  - One other operation (setting either lower or upper)



Best Case

- In the best case the target is the midpoint element of the array
  - Requiring just one iteration of the while loop

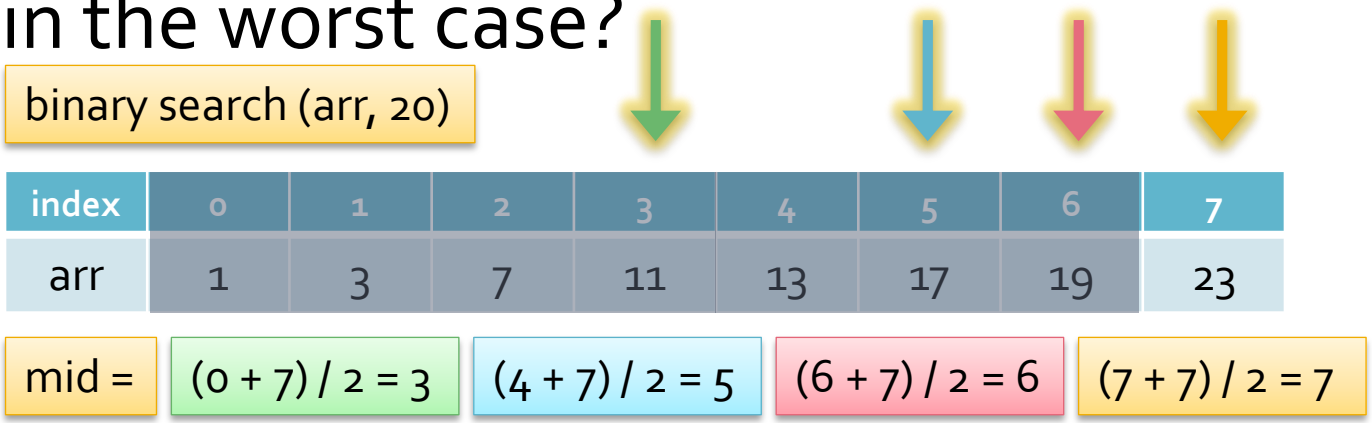
binary search (arr, 11)

index	0	1	2	3	4	5	6	7
arr	1	3	7	11	13	17	19	23

mid = (0 + 7) / 2 = 3

Worst Case

- What is the worst case for binary search?
  - Either the target is not in the array, or
  - It is found when the search space consists of one element
- How many times does the while loop iterate in the worst case?



Analyzing the Worst Case

- Each iteration of the while loop halves the search space
  - For simplicity assume that n is a power of 2
    - So $n = 2^k$ (e.g. if $n = 128$, $k = 7$ or if $n = 8$, $k = 3$)
- How large is the search space?
  - After the first iteration the search space is $n/2$ or $n/2^1$
  - After the second iteration the search space is $n/4$ or $n/2^2$
  - After the k^{th} iteration the search space consists of just one element $n/2^k = n/n = 1$
    - Note that as $n = 2^k$, $k = \log_2 n$
  - The search space of size 1 still needs to be checked
  - Therefore at most $\log_2 n + 1$ iterations of the while loop are made in the worst case

Cost function: $t_{\text{binary search}} = 5(\log_2(n)+1)+4$

Average Case

- Is the average case more like the best case or the worst case?
  - What is the chance that an array element is the target
    - 1/*n* the first time through the loop
    - 1/(*n*/2) the second time through the loop
    - … and so on …
- It is more likely that the target will be found as the search space becomes small
  - That is, when the while loop nears its final iteration
  - We can conclude that the average case is more like the worst case than the best case

Binary Search vs Linear Search

n	Linear Search $(3n+1)/4$	Binary Search $\log_2(n)+1$
10	8	4
100	76	8
1,000	751	11
10,000	7,501	14
100,000	75,001	18
1,000,000	750,001	21
10,000,000	7,500,001	25