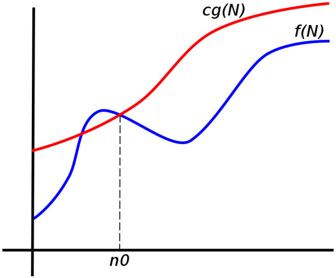
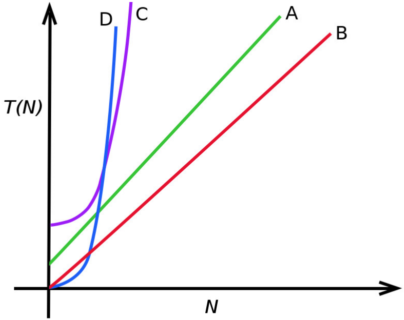
CO580 Algorithms

Dr Timothy Kimber

January 2018



Recall



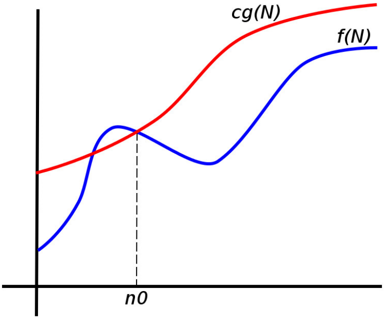
Asymptotic Notation

Algorithm performance is often expressed using asymptotic notation which captures the key ideas we discussed.

- Functions with similar growth are grouped into sets.
- The sets denote a bound on the functions.
- A function f is in
  - $O(g)$ if g is an asymptotic upper bound for f;
  - $\Omega(g)$ if g is an asymptotic lower bound for f;
  - $\Theta(g)$ if g is an asymptotically tight bound for f.

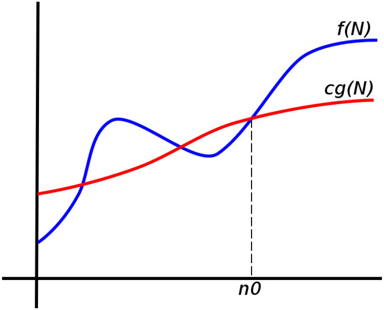
  where g is a characteristic function.
- The definitions of O, Ω and Θ are broad — coefficients are not significant.
- So, (A) and (B) above are both in $O(N)$, but (C) and (D) are not because they grow too fast.

Big O: Upper Bound



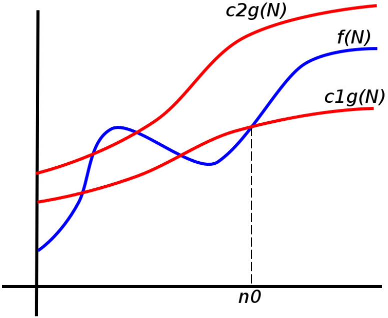
$$O(g(N)) = \left\{ f(N) \;\middle|\; \begin{array}{l} \text{there are positive constants } c \text{ and } n_0 \\ \text{such that } 0 \leq f(N) \leq c\, g(N) \text{ for all } N \geq n_0 \end{array} \right\}$$

Big Omega: Lower Bound



$$\Omega(g(N)) = \left\{ \begin{array}{l|l} f(N) & \text{there are positive constants } c \text{ and } n_0 \\ & \text{such that } 0 \leq c\,g(N) \leq f(N) \text{ for all } N \geq n_0 \end{array} \right\}$$

Big Theta: Tight Bound



$$\Theta(g(N)) = \left\{ f(N) \mid \begin{array}{l} \text{there are positive constants } c_1,\ c_2 \text{ and } n_0 \\ \text{such that} \\ 0 \le c_1\, g(N) \le f(N) \le c_2\, g(N) \text{ for all } N \ge n_0 \end{array} \right\}$$

Asymptotic Notation

Even though $O(N)$ etc. are sets, bounds are usually stated like this:

- $N + 5 = O(N)$
- $T(N) = O(N^2)$
- (rather than $T(N) \in O(N^2)$)

Also, even though asymptotic notation applies to functions, it is (abusively) applied to algorithms too.

- We say "SimpleSearch is $O(N)$"

We use the same notation to talk about other resources:

- We say "the space complexity of MergeSort is $\Theta(N)$"

Space Complexity

The SimpleSearch procedure requires:

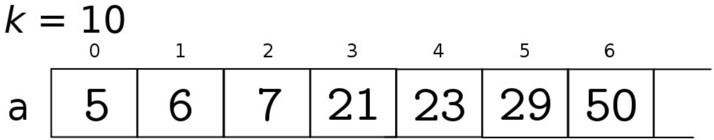
- $\Theta(1)$ space for the best case
- $\Theta(1)$ space for the worst case
- $\Theta(1)$ space for any input

"1" is the normal reference function for any constant

- The space used by the input is ignored
- If not this would mask differences due to algorithm
- SimpleSearch only needs space for a few local variables (e.g. a loop counter). This does not depend on N.

Better Search

- So, we have a $O(N)$ search algorithm. Can you do any better?



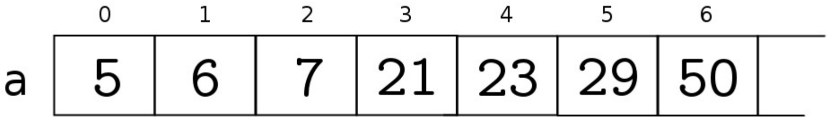
- You have already seen Binary Search.
- It uses the fact that elements are ordered.
- Checking an element in the middle means you can discount half the remaining data.

Binary Search: Design

Question

Binary Search creates regions in a. What properties should the algorithm maintain for it to be correct?

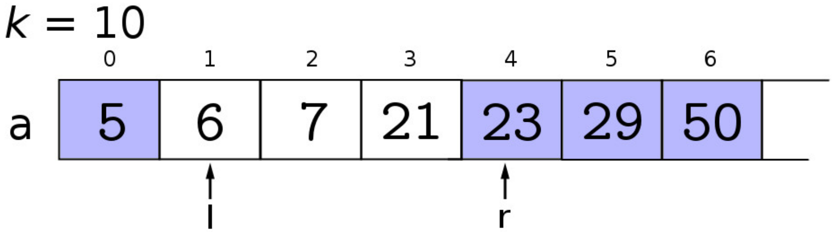
$k = 10$



Loop Invariants: A Design Tool

A loop invariant is a property that is true before every iteration of a loop.

- Used to ensure/prove correctness, also helps in design



In Binary Search we assert that:

- Elements left of index l are known to be less than k;
- Elements at index r or above are known to be greater than k;
- so $a[l, \ldots, r-1]$ is unsearched.

Loop Invariants

A loop invariant must satisfy each of these:

initialisation The invariant must be true before the loop begins

maintenance If the invariant is true before a loop iteration, then it is still true before the next

termination When the loop ends the invariant implies a useful property of the algorithm

A tricky problem can be solved by coming up with an idea for an invariant

- The three conditions help see how (and if) it would work in detail

Loop Invariants

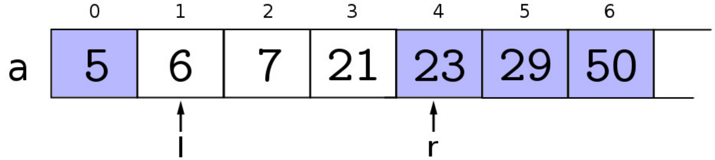
For Binary Search:

initialisation The whole of a should be unsearched, which gives initial values for l and r

maintenance The invariants must hold before each iteration, which gives the form of updates of l and r

termination If the loop ends nothing should be unsearched, which gives the loop condition

$k = 10$



Loop Invariants

- Elements left of l are less than k
- Elements r and above are greater than k
- $a[l, \ldots, r-1]$ is unsearched

Binary Search($a[1, \ldots, N]$, k)

```
l = 1, r = N + 1                       // all unsearched
while l < r                            // more to search
  m = l + (r-l) / 2
  if      (k == a[m]) return True
  else if (k < a[m])  r = m            // search up to m-1
  else                l = m + 1        // search down to m+1
return False
```

Performance

What is the worst case time complexity of Binary Search?

Binary Search($a[1, \ldots, N]$, k)

Code	Cost	Executions
`l = 1, r = N + 1`	c1	1
`while l < r`	c2	??
`  m = l + (r-l) / 2`	c3	??
`  if (k == a[m])`	c4	??
`    return True`	c5	0
`  else if (k < a[m])`	c6	??
`    r = m`	c7	??
`  else`		
`    l = m + 1`	c8	??
`return False`	c9	1

Intuition: loop executes $\log_2 N$ times.

Performance

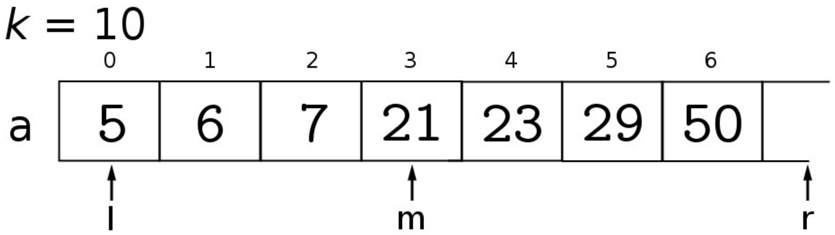
Alternative: analyse the recursive form of the program.

BinSearch(a, l, r, k)

```
                                            Cost
  if (l >= r)                                c1
    return False                             c2
  m = l + (r-l) / 2                          c3
  if (k == a[m])                             c4
    return True                              c5
  else if (k < a[m])                         c6
    return BinSearch(a, l, m, k)             T(N')
  else
    return BinSearch(a, m+1, r, k)           T(N'')
```

- where N' and N'' are numbers left to search
- Exercise: what are N' and N'' in the worst case? Be exact.

Worst Case Recursion



- m is always placed at $1 + \lfloor N/2 \rfloor$
- if N is odd: $N' = N'' = \lfloor N/2 \rfloor$
- if N is even: $N' = \lfloor N/2 \rfloor$, $N'' = \lfloor N/2 \rfloor - 1$
- So the worst case is when $k < a[0]$
  - If $N > 0$, will have $\lfloor N/2 \rfloor$ unsearched elements

Performance

We now have enough information to write a worst case formula for $T(N)$

BinSearch(a, l, r, k)

```
                                         Cost
    if (l >= r)                           c1
      return False                        c2
    m = l + (r-l) / 2                     c3
    if (k == a[m])                        c4
      return True                         c5
    else if (k < a[m])                    c6
      return BinSearch(a, l, m, k)        T(floor(N/2))
    else
      return BinSearch(a, m+1, r, k)      ?
```