Efficiency of algorithms

- Algorithms
- Computational resources: time and space
- Best, worst and average case performance
- How to compare algorithms: machine-independent measure of efficiency
- Growth rate
- Complexity measure O( )

Algorithms

- Algorithm is a well-defined sequence of steps which leads to solving a certain problem.

Steps should be:

- concerete
- unambiguous
- there should be finitely many of them

Efficiency of algorithms

- How much time does it need
- How much memory (space) does it use

Binary search and linear search

- One seems faster than the other
- Can we characterise the difference more precisely?

Best, worst and average case

Linear search:

- Best performance: the item we search for is in the first position; examines one position
- Worst performance: item not in the array or in the last position; examines all positions
- Average performance (given that the item is in the array): examines half of the array

Binary search

- Best case
- Worst case
- Average case

Binary search

- Best case - item in the middle, one check
- Worst case
- Average case

Binary search

- Best case - item in the middle, one check
- Worst case - item in the last possible division; the maximal number of times an array of length N can be divided is $\log_2 N$ ($2^{\log_2 N} = N$).
- Average case

Binary search

- Best case - item in the middle, one check
- Worst case - item in the last possible division; the maximal number of times an array of length N can be divided is $\log_2 N$
- Average case: somewhere in between; ½ $\log_2 N$

Which is more useful?

- For real time programming: the worst case
- For getting a general idea of running time: average case; however, often difficult to establish
- For choosing between several available algorithms: helps to know what **is** the best case (maybe your data are in the best case format, for example random).

How to compare

- Suppose we settle on comparing the worst case performance of linear and binary search.
- Where do we start?
- Timing
- ...

Machine Independence

- The evaluation of efficiency should be as machine independent as possible.
- For the *time complexity* of an algorithm,
  - we count the number of basic operations the algorithm performs
  - we calculate how this number depends on the size of the input.
- *Space complexity*: how much extra space is needed in terms of the space used to represent the input.

Some clarifications

- "Basic operations"?
- "Size of input"?

Example

```
linearSearch(int[] arr, int value){
  for(int i=0; i<arr.length; i++)
    if(arr[i]==value) return true;
  return false;}
```

- Basic operations: comparing two integers; incrementing i. Assume both take time C.
- Size of input: length of the array N.
- Time usage in the worst case: t(N)=N*3C.

Binary search

```
binarySearch(int[] arr, int value){
   int left = 0;
   int right = arr.length - 1;
   int middle;
   while (right >= left) {
      middle  =  (left+right)/2;
      if (value == arr[middle]) return true;
      if (value < arr[middle]) right=middle-1;
      else left  =  middle+1;
      }
      return false;
}
```

Analysis of binary search

- Size of input = size of the array, say N
- Basic operations: assignments and comparisons
- Total number of steps: 3 assignments plus a block of assignment, check and assignment repeated $\log_2 N$ times.
- Total time = $3C + 5C' \log_2 N$

Rate of Growth

We don't know how long the steps actually take; we only know it is some constant time. We can just lump all constants together and forget about them.

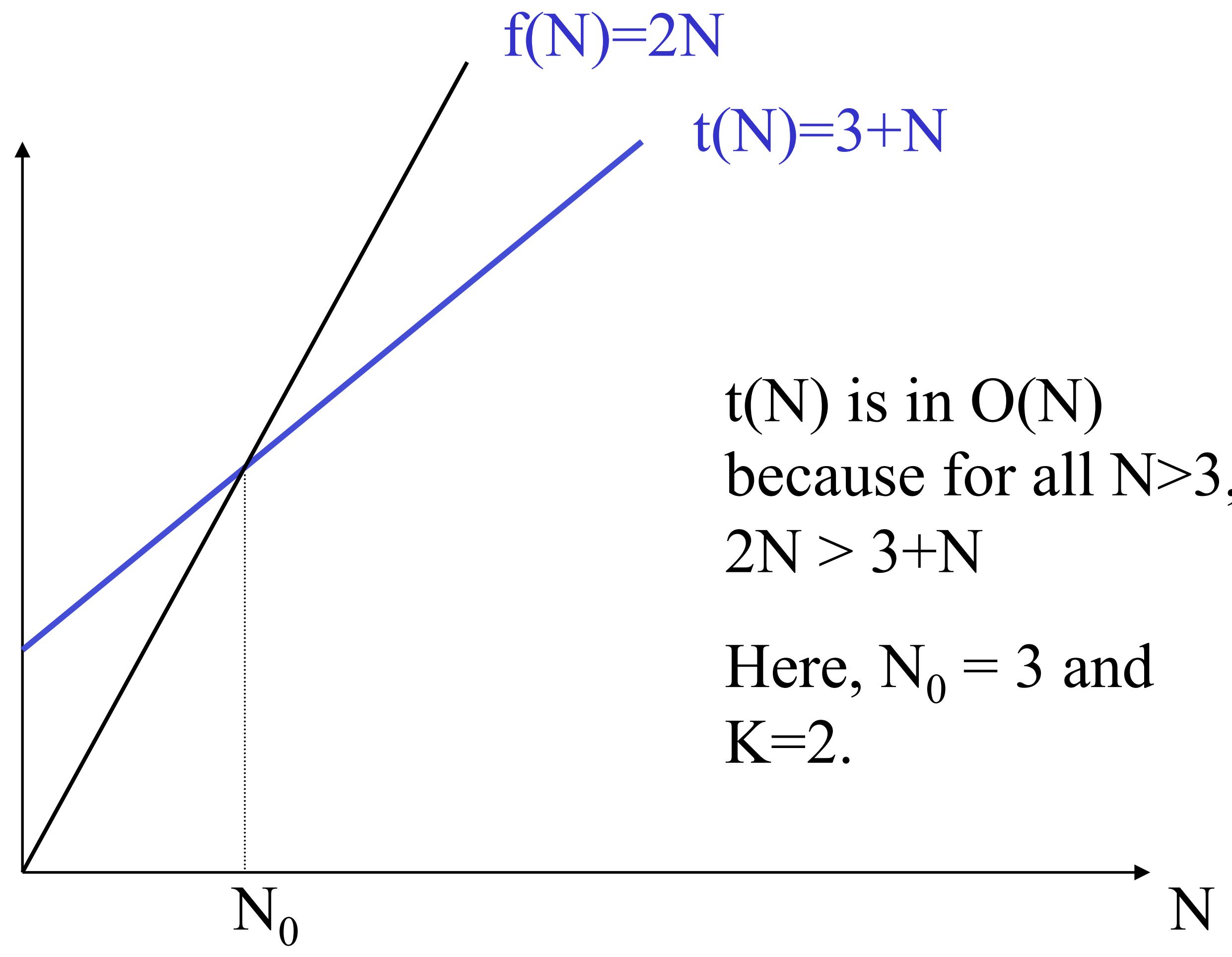
What we are left with is the fact that the time in sequential search grows linearly with the input, while in binary search it grows logarithmically - much slower.

O() complexity measure

Big O notation gives an asymptotic upper bound on the actual function which describes time/memory usage of the algorithm.

The complexity of an algorithm is O(f(N)) if there exists a constant factor K and an input size N_0 such that the actual usage of time/memory by the algorithm on inputs greater than N_0 is always less than K f(N).

Upper bound example



t(N) is in O(N) because for all N>3, 2N > 3+N

Here, $N_0 = 3$ and K=2.

In other words

An algorithm actually make g(N) steps,

for example $C + C'\log_2 N$

there is an input size N' and

there is a constant K, such that

for all N>= N' , g(N) <= K f(N)

then the algorithm is O(f(N)

Binary search is O(log N)

Comments

Obviously lots of functions form an upper bound, we try to find the closest

We also want it to be a simple function, such as

constant O(1)

logarithmic O(log N)

linear O(N)

quadratic, cubic, exponential...

Typical complexity classes

Algorithms which have the same O( ) complexity belong to the same *complexity class*.

Common complexity classes:

- O(1) constant time: independent of input length
- O(log N) logarithmic: usually results from splitting the task into smaller tasks, where the size of the task is reduced by a constant fraction
- O(N) linear: usually results when a given constant amount of processing is carried out on each element in the input.

Contd.

- O(N log N) : splitting into subtasks and combining the results later
- O(N^2) quadratic: usually arises when all pairs of input elements need to be processed
- O(2^N) exponential: usually emerges from a brute-force solution to a problem (exhaustive enumeration).

Practical hints

- Find the actual function which shows how the time/memory usage grows depending on the input N.
- Omit all constant factors.
- If the function contains different powers of N, (e.g. $N^4 + N^3 + N^2$), leave only the highest power (N^4).
- Similarly, an exponential (2^N) eventually outgrows any polynomial in N.

Warning about O-notation

- O-notation only gives sensible comparisons of algorithms when N is large
  Consider two algorithms for same task:
  Linear: $f(N) = 1000\ N$
  Quadratic: $f'(N) = N^2/1000$
  The quadratic one is faster for $N < 1\ 000\ 000$.
- Some constant factors are machine dependent, but others are a property of the algorithm itself.

Summary

- Big O notation is a rough measure of how the time/memory usage grows as the input size increases.
- Big O notation gives a machine-independent measure of efficiency which allows comparison of algorithms.
- It makes more sense for large input sizes. It disregards all constant factors, even those intrinsic to the algorithm.