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:

- concrete
- 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 - 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: item is found after performing half of the possible number of divisions; $\frac{1}{2} \log_2 N$

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"?

Size of input

- It is up to us to decide what is a useful parameter to vary when we measure efficiency of an algorithm.
- For algorithms searching a linear collection, the natural choice for size of input is the *number of items* in the collection which we are searching (e.g. length of the array).

Size of input contd.

- Graph search: we may be interested in how time grows when the *number of nodes* increases, or in how time grows when the *number of edges* increases.
- Sometimes we measure efficiency as a function of several parameters: e.g. number nodes *and* edges in a graph.

Basic operations

- Basic operations are operations which take constant time (at most time C for some constant C).
- In other words, time required to perform the operation does not grow with the size of the operands, or is bounded by a constant.

Basic operations contd.

- If we are not sure how operations are implemented, we have to exercise judgement: can something be in principle implemented as a constant time operation?
- For example, adding two 32-bit integers can be, but adding a list of 32-bit integers can not: it is going to take longer for a longer list.

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. Constant time (at most some C) spent at each iteration.
- Size of input: length of the array N.
- Time usage in the worst case: t(N)=C * N.

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. Assume 3 assignments take at most time C_1 and at each iteration we spend at most time C_2.
- Total time = $C_1 + C_2 \log_2 N$

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.

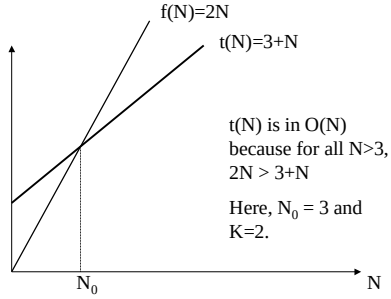
What we are left with is the fact that the time in sequential search grows linearly with the input, while in binary search it grows logarithmically - much slower.

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



In other words

An algorithm actually makes g(N) steps,

for example $g(N) = C_1 + C_2 \log_2 N$

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 in O(f(N)).

Binary search is O(log N):

$C_1 + C_2 \log_2 N <= (C_1 + C_2) \log_2 N$ for N > 2

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.

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: $g(N) = 1000\ N$ is in $O(N)$
  Quadratic: $g'(N) = N^2/1000$ is in $O(N^2)$
- 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.

Recommended reading

- Shaffer, Chapter 3 (note that we are not going to use Ω and Θ notation in this course, only the upper bound O()).

Informal coursework

Which statements below are true?

- If an algorithm has time complexity $O(N^2)$, it always makes precisely N^2 steps, where N is the size of the input.
- An algorithm with time complexity O(N) is always runs slower than an algorithm with time complexity $O(\log_2(N))$, for any input.
- An algorithm which makes $C_1 \log_2(N)$ steps and an algorithm which makes $C_2 \log_4(N)$ steps belong to the same complexity class (C_1 and C_2 are constants).