Lab 4: Undirected graphs, running time intuitions

Data Structures and Algorithms for CL III
15 Nov 2019

Getting better intuitions for the running time of algorithms

- Empirical testing is easy and often effective
  - But your computing environment and so on may vary
  - More importantly, empirical testing doesn't necessarily indicate why a program is slow or how to fix it
- But it takes time to get an intuition for analysis, so let's discuss some examples and a general approach

An angle of attack

1. Count individual distinct instructions
2. Identify loops (recursive or iterative)
3. Count the length of loops (5, n, indefinite)
   3.1. Things like while loops might seem indefinite, but note the loop end condition if any
4. In simple cases, just multiply the instructions times any containing loops' lengths

Note: Remember time complexity is measured asymptotically. I.e. if the exact running time is $2n^2 + 3n + 128$, the n^2 will dominate over its constant coefficient 2 and the other terms $3n$ and 128 as n grows, so we say it is $O(n^2)$.

This means we don't need to count every instruction, just find the biggest performance culprits.

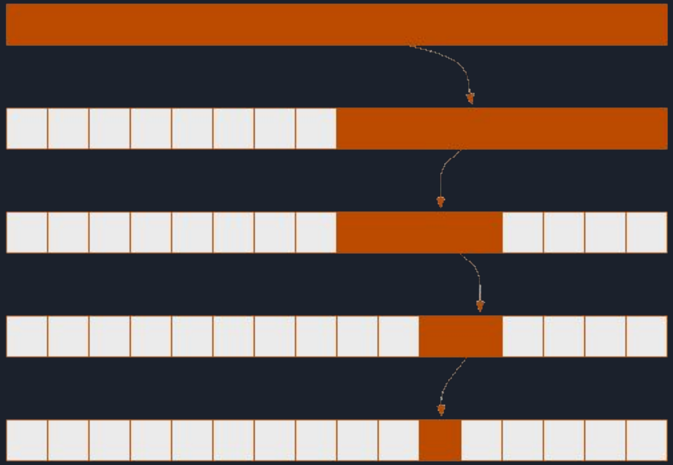
A few heuristics for time complexity

- Is it a fixed loop independent of the input?
  - Constant time, i.e. $O(1)$
- Is it a single loop of input size *n*?
  - Linear time, i.e. $O(n)$
- Nested loops of input size *n*?
  - Quadratic time, e.g. $O(n^2)$, $O(n^3)$, $O(n^4)$, etc.
- Are you trying every permutation of the input of size *n*?
  - Factorial time, i.e. $O(n!)$

A few trickier heuristics for time complexity

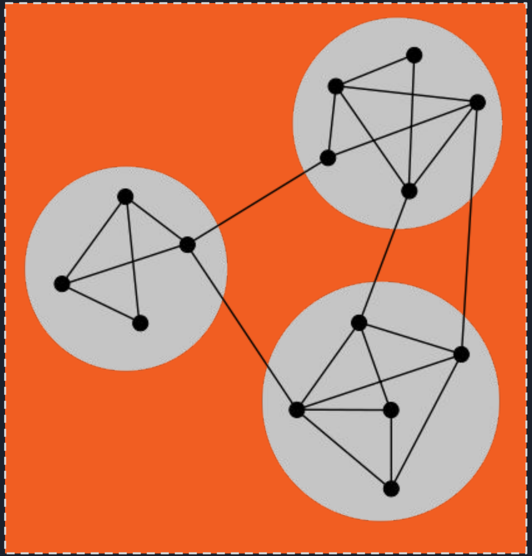
- Does it resemble binary search?
  - By breaking the input into progressively smaller pieces, each takes less time than the last to process
  - Typically logarithmic time, i.e. $O(\log(n))$
- Is it a divide-and-conquer style sorting algorithm like Mergesort, Quicksort, Heapsort?
  - Involves doing some binary-search like operation n times
  - Often so-called quasilinear time, e.g. $O(n\log(n))$

```
def bs(A, i, val):
    if i = 1:
        if A[0] = val:
            return true
        else:
            return false
    if val < A[i / 2]:
        return bs(A[ 0...( i / 2 - 1 ) ], i / 2 - 1, val)
    else if val > A[i / 2]:
        return bs(A[ ( i / 2 + 1 )...i ], i / 2 - 1, val)
    else:
        return true
```



Binary search pseudocode and diagram of its execution on an array. In Mergesort for example we do this kind of search *n* times, once for each element, thus O(*n*log(*n*)) time.

Motivation: How resilient is the internet to shark attacks?



- Submarine communications cables are sometimes subject to sharks biting them, causing failures
  - Are we safe? How many possible points of failure are there?
- How do we know not just **if** two vertices are connected, but **how** connected they are?
- What follows is a total aside, but it will help motivate why connected components are a useful idea

Motivation: more on connectivity -- a linguistics example

- Suppose we have a set of words and a thesaurus
    - The words are all synonyms of each other by some *d* degrees of separation (e.g. a connected graph)
    - How closely related are those words to each other?
    - This problem could actually be viewed as the same problem as the sharks

Motivation: more on connectivity

- We can measure how related a set of words are or how resilient the internet is to sharks by something called **connectivity**
  - A vertex cut is any set of vertices in a graph which if you took them out, would separate it into 1 or more distinct connected components
  - The connectivity of a graph is the size of the smallest vertex cut for that graph
  - A graph is 1-connected if you only have to take out 1 vertex to disconnect it, 2-connected if at least 2, in general k-connected if the size of the smallest vertex cut is k
- There are other measures of graph/network inter-connectivity, even things as simple as average degree
- But this one puts to use what we already know about connected components to find not just how a graph is inter-connected, but where