CSCI 262 Lecture 7 – Selection Sort & Analysis of Algorithms

Outline

- Selection Sort – an algorithm for putting elements in a list into order
- How do computer scientists measure work?
  - Count some basic operations
  - What does our count look like, generalized to inputs of size *n*?
- Big O, a little more formally than before
  - Primary concern: how does cost grow as *n* grows (where *n* is some statistic of the input)
  - Ignore lower order terms (functions that grow more slowly) and constants
  - Express in simplest terms – e.g., $O(n^2)$, not $O(2n^2 + 3n + 1)$

Readings

Start reading Chapter 11 in your textbook

More about Moore's Law (it doesn't say what most people think):

https://en.wikipedia.org/wiki/Moore%27s_law

Self Check

1. What is the closed-form expression for the sum of integers from 0 to n ($\sum_{i=0}^{n} i$)?
2. Using the simplification rules we learned in class, what is the above expression in "Big O"?
3. Which grows faster?
   - n or log(n)
   - n^3 or 2^n
   - 2^n or n!

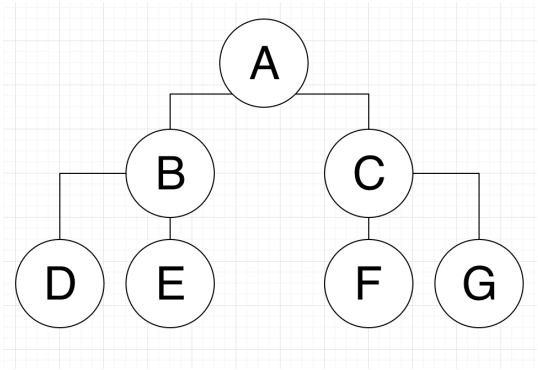
For Further Practice

Computer scientists and mathematicians are extremely interested in the boundary between *polynomial* algorithms (with complexities of the form $O(n^p)$) and *exponential* algorithms ($O(b^n)$). One reason for this is because there are categories of problems for which the best known algorithms are exponential, where no proof has been found that we cannot do better (this is the famous P = NP problem). Another reason is simply that exponential algorithms are generally hopelessly intractable – there is no effective way of solving problems of useful size in many cases.

Why can't we just wait for computers to get faster? Suppose we take Moore's Law as suggesting computers get twice as "fast" every two years (this isn't what Moore's Law says, but never mind):

1. If an $O(2^n)$ algorithm takes 1 hour to solve for an input of size *k*, what size input can we solve in an hour with a computer that is twice as fast?
2. Same question, but for an $O(n^2)$ algorithm.

Graph



Depth First Search

Find a path from A to F with DFS (using a stack)

Breadth First Search

Find a path from A to F with BFS (using a queue)