Introduction to Algorithms and Complexity Theory

Iris Cong
UCLA Math Circle - HS II

Example 1. Algorithm to find the maximum of a list of N positive integers (call this list $L_0, \ldots L_{N-1}$):

```
initialize accumulator M = 0
for each j = 0, 1, ... N-1 in order:
     if L_j > M
          set M = L_j
return M
```

How can we characterize the running time of an algorithm? To do this, we introduce the concept of what is known as *big-O notation.*

Definition 1. $f(n) = O(g(n))$ if there exist constants $c > 0$ and $N > 0$ such that for all $n \geq N$ we have

$$f(n) \leq cg(n) \tag{1}$$

Example 2: Some examples of f, g and c satisfying Eq. (1):

1. $f(n) = n$, $g(n) = 3n$ (works with $c = 3$)
2. $f(n) = n + 10000000$, $g(n) = 4n$ (works with $c = 5$)
3. $f(n) = n^2 + 99999n + 300$, $g(n) = n^3$ (works with $c = 2$)
4. $f(n) = 1000n^2$, $g(n) = e^n$ (works with $c = 1$)
5. $f(n) = \log n$, $g(n) = n$ (works with $c = 1$)

Warm-up 1. Consider the following algorithm on a List of integers of length N:

```
define f1(List): // line 1
     for each j = 0, 1, ... N-1 in order: // line 2
          for each k = 0, 1, ... N-1 in order: // line 3
               if List_j = List_k return true // line 4
     return false
```

We've demonstrated together that f1 is $O(N^2)$. As another warm-up, what does f1 actually do?

Warm-up 2. What does the following algorithm do? What is its runtime in big-O notation, if List has length N?

```
define f2(X, List)
     initialize A = 0
     for each j = 0, 1, ... N-1 in order:
          if List_j = X set A = A+1
     return A
```

Warm-up 3. What does the following algorithm do? What is its runtime in big-O notation, if List has length N?

```
define f3(List)
     initialize B = 0
     for each j = 0, 1, ... N-1 in order:
          for each k = 0, 1, ... j in order:
               if List_k > List_j set B = B+1
     return B
```

Now that we've taken a good look at some sample algorithms, let's roll up our sleeves and come up with some ourselves!

Problem 1a. Given a list L of integers, provide an algorithm in the style of Example 1 to find the contiguous subsequence of the list that has the maximal sum. An example input/output is shown below:

31	-41	59	26	-53	58	97	-93	-23	84

Using big-O notation, determine the runtime of your algorithm.

b. Can you rewrite your algorithm from part (a) to have better asymptotic runtime?

c. Is the algorithm you provided in (b) optimal? If so, give a short explanation; if not, try to come up with a faster one.

Problem 2. Consider the simple problem statement of sorting a list of N integers. Without turning to the next page, write an algorithm that sorts the list in nondecreasing order. What is the time complexity of your sort algorithm, using Big-O notation in terms of N?

Problem 3. *The MergeSort algorithm*.

In the previous problem, you came up with an algorithm that could sort a list of N numbers -- hopefully, it was at least as fast as $O(N^2)$. In this problem, we'll work together to write a sorting algorithm that can run in time $O(N \log N)$. (If you already got this in Problem 2, you may skip ahead to the next problem). Here's a simpler problem to get you started:

Suppose you are given two lists of numbers A and B, and that you have some additional information: A and B are both already sorted in nondecreasing order. Write an algorithm to construct a new list C, which has all the elements of A and all the elements of B, but is also sorted. For instance, if

A = | 1 | 3 | 5 | 7 |

B = | 2 | 4 | 6 | 8 |

→ C = | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

a. Suppose A has length N and B has length M. What is the time complexity of your algorithm in Big-O? How about the special case in which A and B both have length N?

b. Can you use your solution to part (a) to come up with a sorting algorithm? What is the time complexity of this sorting algorithm?

Problem 4. The *interval scheduling problem* is defined as follows: Input will be given as a set of tasks, each represented by an interval in which it must be executed. For example, in Figure 1, Task A must run from time 0-6, task 2 must run from 1-4, etc. As in Figure 1, we see that not all tasks can be executed, since a lot of them overlap, but you'd like to finish as many of them as possible. Your algorithm's job is to find the *maximal subset* of the tasks that can all be run without overlap. In Fig. 1, the maximal subset would be {B, E, H}, since that is the only way we can schedule more than 2 tasks.

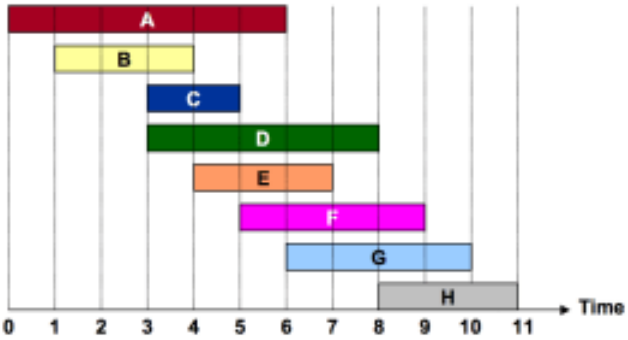
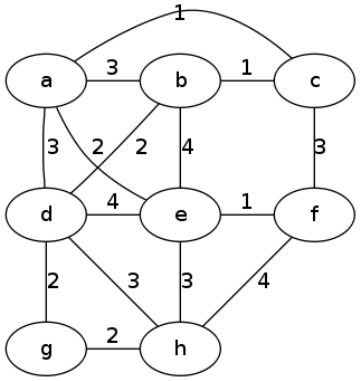


Figure 1: Interval scheduling problem

a. Write an algorithm to solve the interval scheduling problem. What is its complexity in terms of N, using Big-O notation, where N is the number of intervals given?

b. Is the complexity you found in part a) polynomial in N? If not, can you come up with a better algorithm?

Problem 5. A *graph* is a collection of vertices and edges between vertices. On a given graph, we can associate nonnegative *edge costs* to each of the edge: for instance, this could represent miles between cities on a map, or communication costs between villages, etc. A *path* from a vertex A to another vertex B is simply an alternating sequence ($A = v_0, e_1, v_1, \ldots e_k, v_k = B$) of vertices and edges such that e_j joins vertices v_{j-1} and v_j for each j. The *cost* or *length* of a path would be the sum of all the costs of the edges e_j.



Give an algorithm to find the shortest path from a starting vertex A to a finish vertex B in a given graph. Express the running time of your algorithm using big-O notation in terms of the number of vertices V and the number of edges E in the graph. Try to make your algorithm optimal, or at least polynomial in V and E.

Challenge Problems:

1. The algorithm given in Warm-up 2 can be improved in terms of asymptotic runtime. Can you write a faster algorithm with the same functionality?

2. In Problem 5, you were asked to come up with a polynomial-time algorithm to find the shortest path between two vertices in a graph. Will your algorithm work if the edge costs can be negative? Alternatively, can you come up with a similar algorithm to find the *longest* path between the two vertices? Why or why not?