CS 130A: Introduction and Admin Stuff

Subhash Suri

January 10, 2019

1 Data Structures and Algorithms

- Niklaus Wirth famously said that *Data Structures + Algorithms = Programs*.
- Today's computing world is more diverse and complex than in Wirth's time but a vast amount of computer science is still primarily about organizing data and designing algorithms.
- What is a data structure anyway? According to Wiki,

  > *It is a data organization, management and storage format that enables efficient access and modification.*

- In other words, it's an agreement about (1) how to store objects in memory, (2) what data operations to support, (3) the algorithms for those operations.
- You have already seen many elementary data structures, such as linked lists, queues, arrays, trees, perhaps even heaps and hash tables.
- In this course, we will cover Hash Table, Heaps, Balanced Binary Trees, Splay Trees, B-Trees, and some structures and algorithms for graphs.
- Unlike your previous courses, a major focus for us will be *rigorous theoretical analysis* of these data structures, so you will have a better appreciation of "why" they work, not just knowledge of what the code looks like.
- In fact, we will see that the *theory* behind simple data structures such as hash tables is far from simple, and utilizes interesting concepts from probability theory and discrete math. Only when we engage in that analysis we develop a true appreciation of the underlying beauty of these data structure.
- A common theme of most data structures we study is the following:

1. We have a set of n objects. These can be customer records, names of people, DNA sequences, music files, or images.
2. We want to organize them in a "searchable" structure, which will allow us to "quickly" find the object we are looking for.
3. We typically also want to modify these structures as new objects are added to the set, or old ones deleted.
4. For instance,
   (a) The Google "search" function is essentially a data structure problem: objects are the documents (web pages) organized by their keywords.
   (b) The operating system of your computer needs to maintain a data structure that lets it quickly find the memory location of any variable in your program.
   (c) Firewall systems at routers need to decide for each incoming data packet whether to block it or to allow it, etc.
   (d) Besides these stand alone applications, virtually every algorithm depends on thoughtful use of appropriate data structures for its *efficient design*. Dijkstra's shortest paths, Prim's minimum spanning tree, Huffman coding, to name a few.
5. We want our data structures to be *efficient*: search should be fast, and the data structure should not take too much memory.
6. An ideal search time would be $O(1)$, regardless of the size of the data n, which is sometimes, but not always, achievable. But search times of the order of $\log n$ are highly desirable. They grow very slowly with n. In some cases, one has to be content with search times of order $\sqrt{n}$; but there are difficult search problems where beating the naive bound of $O(n)$, even slightly, has been a challenge.
7. For the purpose of general purpose design of data structures, we assume that the objects in our set have simple identifiers, say, integer valued keys. In fact, one of the first data structures, hash table, helps us convert arbitrarily long objects into small-size numeric keys.

- We will design and analyze our data structures and algorithms under the *asymptotic* complexity model: focus on the (asymptotic) *growth rate* as a function of input size n, ignoring constant factors and lower order terms.
- We want our bounds to hold in *worst-case*, so that the data structures and algorithms perform well even if the input is given by an *adversary*.
- These are concepts you are familiar with, but you should review.

- The most common asymptotic notation we use is Big-Oh. If an algorithm's time complexity is $T(n) = O(f(n))$, say, $O(n^2)$, it means that the running time is always "bounded above by" some constant times $f(n)$.

- More precisely, there exist constants c and N_0, such that $T(n) \leq c \cdot f(n)$, for all $n > N_0$.

- This notation lets us focus on the long range "growth rate" of the algorithm, say n^2. Once the input gets big enough, the run time is always less than some constant times $f(n)$. The multiplier c allows us to shift $f(n)$ by a constant factor to account for machine speeds, or low level differences in programming languages.

- The terminology was introduced by Paul Bachmann in 1894, but imported to CS and popularized by Don Knuth. (See Dec 17, 2018, NYT article on Knuth "Yoda of Silicon Valley".)

- For data structures, we typically want the run time to be *sub-linear*, such as $O(\log n)$, $O(\sqrt{n})$ etc, which beats the linear scan for each operation.

- The notation Big-Omega $\Omega(f(n))$ is used for "lower bounds" to put a floor on the complexity. Specifically, if $T(n) = \Omega(f(n))$, then there exist constants $c > 0$ and N_0, such that $T(n) \geq cf(n)$, whenever $n > N_0$.

- The notation Big-Theta means "bounded above and below" by the same functions. So, $T(n) = \Theta(f(n))$ means that $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$.

- Some examples.

  1. $T(n) = 2n^3 + 5n^2$.
  2. It is correct to say that $T(n) = O(n^{10})$, but that's a very *loose* upper bound. Similarly, we can also say loosely that $T(n) = \Omega(n)$.
  3. We always want to give the tightest possible bounds, which in this case are $T(n) = O(n^3)$ or $T(n) = \Theta(n^3)$.

- Another useful CS 40 material worth reviewing is "series summation." You will need them frequently for analyzing data structure's performance.

  1. $\sum_{i=1}^{n} i = \Theta(n^2)$.
  2. $\sum_{i=1}^{n} i^2 = \Theta(n^3)$.
  3. $\sum_{i=1}^{n} i^r = \Theta(n^{r+1})$.
  4. $\sum_{i=1}^{k} n^i = \Theta(n^k)$.

5. $\sum_{i=1}^{k} 1/i = \Theta(\ln k)$.
6. $n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$.

- Instead of reviewing any further material on asymptotic analysis and algorithm design, let us engage in a problem solving exercise.

2 Max Subsequence Problem: Four Algorithms

- A sequence is an *ordered* list. We are given a sequence $S = (a_1, a_2, \ldots, a_n)$ of numbers (integers), which can be positive or negative.
- A *subsequence* of S is a *contiguous block*, such as $a_i, a_{i+1}, \ldots, a_j$.
- The *value* of a subsequence is the *sum* of all the numbers in it, namely, $\sum_{k=i}^{j} a_k$.
- For instance, $S = (4, 3, -8, 2, 6, -4, 2, 8, 6, -5, 8, -2, 7, -9, 4, -1, 5)$ is a sequence. $(3, -8, 2, 6)$ is subsequence of value 3..
- The max subsequence problem is to find the subsequence of S with the maximum possible value. (If all numbers are negative, we return $S = \emptyset$ of zero value.)
- *If all numbers in the input sequence are negative, then we simply return the empty subsequence, with value* 0*, as the answer.*
- While not a data structure problems, it is an excellent pedagogical exercise for design, correctness, and asymptotic runtime analysis of algorithms.
- We will discuss four different algorithms, resp., of time complexity $O(n^3)$, $O(n^2)$, $O(n \log n)$ and $O(n)$. The last one is clearly optimal, since any algorithm must scan the entire input.
- Even on modern computers, algorithm 1 may take days to months for $n = 10^6$, while algorithm 4 will finish in fraction of a second.

2.1 $O(n^3)$ Algorithm

- Input is a sequence $S = (a_1, a_2, \ldots, a_n)$. The goal is to find the maximum value subsequence of S.
- The cubic algorithm is simple: just compute the values of all possible subsequences, and return the best. Each subsequence is defined by its unique starting and ending indices i and j.

1. Init. maxSum $= 0$; $I = J = 0$
2. for $i = 1$ to n
3. for $j = i$ to n
   (a) initialize thisSum $= 0$
   (b) for $k = i$ to j
   (c) thisSum $+ = a_k$
   (d) if thisSum $>$ maxSum
   (e) maxSum $=$ thisSum
   (f) $I = i$; $J = j$
4. return maxSum, I, J.

- Correctness. The algorithm is clearly correct because it checks all possible subsequences of S, trying all possible start points i and end points j. The innermost loop keeps track of the largest value subsequence.

- Time Complexity. The total number of steps can be written as the following nested sum:

$$O\left(\sum_{i=1}^{n}\sum_{j=i}^{n}(j-i)\right)$$

- The summations handle the two nested loops, and the innermost term $(j - i)$ is the cost of summing the terms from a_i to a_j.

- The remaining steps are $O(1)$ each.

- We can compute this sum exactly using our arithmetic series formulas, but we can also get a quick and dirty estimate, which gives us the correct asymptotic bound: the innermost term $(j - i)$ is at most n, and each of the two summations are over n terms each, so the asymptotic complexity is $O(n^3)$. The true sum is about $n^3/6$, and since the constant is absorbed by big-Oh, we have the correct bound.

2.2 $O(n^2)$ Algorithm

- We can improve the previous algorithm by a factor of n by realizing that computing each $a_i, a_{i+1}, \ldots, a_j$ sum *from scratch* is inefficient.

- The preceding execution of the loop computes the sum of $a_i, a_{i+1}, \ldots, a_{j-1}$, and so we can extend that sum to j in just $O(1)$ time.

- In particular, we can rewrite the previous algorithm as follows: (The variables I, J keep track of the start and end of the current best max subsequence.)

  1. Init. maxSum $= 0$; $I = J = 0$
  2. for $i = 1$ to n
  3. initialize thisSum $= 0$
  4. for $j = i$ to n
     (a) thisSum $+= a_k$
     (b) if thisSum $>$ maxSum
     (c) maxSum $=$ thisSum
     (d) $I = i$; $J = j$
  5. return maxSum, I, J.

- In this algorithm, the algorithm fixes a start index i, and then evaluates all subsequences that start at i and end at $j = i, i+1, \ldots, n$, computing their sum at $O()1)$ each.

- The time complexity of this algorithm, therefore, is $O(n^2)$.

2.3 $O(n \log n)$ Algorithm

- We now explore a divide-and-conquer based algorithm for the max subsequence.

- Suppose we split the input sequence at midpoint $a_{n/2}$.

- What can we say about the max subsequence of S? There are only three possibilities:

  1. It lies entirely in the left half, namely, in $a_1, a_2, \ldots, a_{n/2}$.
  2. It lies entirely in the right half, namely, in $a_{n/2+1}, \ldots, a_n$.
  3. Or, it straddles the midpoint.

- For example, suppose $S = (4, -3, 5, -2, -1, 2, 6, -2)$. Then, $(4, -3, 5)$ is the max subseq in left half, $(2, 6)$ is the max subseq in right half.

- But the straddling sequence $(4, -3, \ldots, 6)$ is the global max subsequence for S, with value 11.

- We find the max subsequences in left and right haves by recursion.

- The key observation is the following:

1. The left half of the straddling sequence is the max subsequence ending with -2 (namely, $a_{n/2}$)..
2. The right half is the max subsequence beginning with -1 (namely, $a_{n/2+1}$)..
3. Call these *anchored* subsequences: one of their endpoints is fixed, and we just need to find the other endpoint.
4. We can find the free (other) endpoint by a simple linear scan: starting at $a_{n/2}$, keep track of the left subsequence sum, and finally return the one with max value. Similarly, for the right anchored subseq.
5. Thus, the straddling max subseq. can be found in $O(n)$ time.

- The time complexity of this divide-and-conquer algorithm is best analyzed through a recurrence:

$$T(n) \quad = \quad 2T(n/2) + O(n)$$

with the boundary condition $T(1) = 1$.

- This is a famous recurrence that solves to $T(n) = O(n \log n)$.

2.4 $O(n)$ Algorithm

- Our final solution is an optimal $O(n)$ time algorithm.
- The *mysterious* algorithm has the following structure.

1. Init. maxSum $= 0$; thisSum $= 0$; $I = J = 0$;
2. for $j = 1$ to n
3. thisSum $+ = a_j$
4. if thisSum $>$ maxSum
5. maxSum $=$ thisSum; $J = j$;
6. else if thisSum < 0
7. thisSum $= 0$; $I = j + 1$;
8. return maxSum, I, J.

- By inspection, the algorithm clearly runs in $O(n)$ time since it just makes one scan over the input sequence.
- But what exactly is it doing? Why does it work?

- In order to understand the algorithm, or more importantly *to discover such an algorithm*, we need to go deeper into the structure of max subsequence, something our less efficient (and more crude) algorithms didn't require.

- This is an important lesson in algorithm and data structure design: more elegant and efficient solutions will often reveal themselves with a deeper understanding of the problem's structure.

- A simple but important observation for the max subseq problem is following: *max subseq never begins or ends with a negative number*. Why?

- A slightly less obvious but more useful observation is that the max subseq also cannot have a *negative value prefix*. Why? Because if it had a negative prefix, we can clip it off and improve the value of the sequence.

- Suppose we are incrementally evaluating subsequences with a fixed start point a_i, and at some point a_j, value of the subseq. $a_i, \ldots, a_j$ becomes < 0.

- Then, we can advance the start point from a_i to a_{j+1}. The reasoning goes as follows:

- First, it is easy to see that the max subseq will never contain the prefix $a_i, \ldots, a_j$. Otherwise we can improve the value of the subseq by simply chopping off the part (with negative value) from a_i to a_j.

- Second, we argue that the max subseq also cannot *start* at any index p between i and j. For the sake of a contradiction, suppose that the max subseq. S^* did start at some p, where $i < p < j$.

- Now, since a_j is the *first* index after i at which the sum from a_i to a_j is negative, the sum from a_i to a_k is > 0, for each $i \leq k < j$. Thus, if the max subseq. starts at a_p, prepending it with the prefix $a_i, \ldots, a_{p-1}$ (of positive value) would make it even larger, contradicting the optimality of S^*.

- Therefore, the algorithm simply keeps adding next term a_j to thisSum, and updating maxSum, until it finds an index at which thisSum < 0. At that point, it simply advances the *start* point of thisSum to $j + 1$, and resets thisSum to 0.

3 Review of Probability Theory

- Finally, we will also depend on our basic knowledge of probability theory, especially for understanding hash functions.

- You should review your PSTAT notes for probability space, random variables, expectation etc.
- I will do a very quick review during hash table discussion, but it will assume prior knowledge and only serve to recall important definitions not to teach them.