6.006最大的收获在于对于数据结构和算法有了比较底层的了解，从而在解决问题的过程中更清楚算法的效率和可扩展性。

* BIG PICTURE *

ADT / DS

ADT (ABSTRACT DATA TYPE)

- Priority Queue
  - A data structure implementing a set S of elements, each associated with a key
  - $O(\log n)$: insert, extract_min, update_key (Heap)
  - $O(1)$: min (Heap)
- L8: Dictionary
  - Maintain a set of items, each with a key
  - $O(1)$: insert, delete, search
- L9: Rolling Hash
  - Maintain string x
  - $O(1)$: r.append(c), r.skip(c)
- L13: Graph

    - $G(V, E)$
    - v.adjacency(), e.weight()

- L4: ADT是由其上可执行的操作以及这些操作的效果与代价所间接定义的。

DS (DATA STRUCTURE)

PRIORITY QUEUE ADT	**Heap**	**Balanced BST (AVL)**
insert	$O(\lg n)$	$O(\lg n)$
min / max	$O(1)$ or $O(n)$	$O(\lg n) \overset{augment}{\rightarrow} O(1)$
PREDECESSOR ADT		
next_larger / smaller	$O(n)$	$O(\lg n)$

RI (Representation Invariant)

- R5: The salient property of a data structure is its **representation invariant (RI)**, which specifies how information is stored.
- L6: 本讲以AVL树为例，首先定义了其不变量：任意节点的子树高度差不大于1；进而证明了其高度是 $\Theta(\log n)$ 。

AUGMENTATION

- L5: 数据结构的增强，从而在现有数据结构上增加更多的低复杂度操作

ALGORITHM

ALGORITHM

SORT	Insertion Sort (Binary)	Merge Sort	Heap Sort	Tree Sort	Counting / Radix Sort
Complexity	$O(n^2)$ swap / $O(n \lg n)$ comp	$O(n \lg n)$	$O(n \lg n)$	$O(n \lg n)$	$O(n)$
Stable*	Yes	Could	No	Yes	Yes
In-place	Yes	No	Yes	No	
Model	Comparison	Comp	Comp	Comp	RAM
名称	插入	归并	堆	树	计数 / 基数

*A sorting algorithm is stable if elements with the same key appear in the output array in the same order as they do in the input array

HASHING	Chaining	Open Addressing	String Matching
	Division	Linear Probing	Karp-Rabin
	Multiplication	Double Hashing	Rolling Hash
	Universal		

NUMERICS	Multiplication	Division	Root
	Karatsuba	Newton's	Newton's
Complexity	$O(n^{\log_2 3})$	same as mul	same as mul

SEARCH	BFS	DFS	Weighted DAG	Dijkstra	Bellman-Ford
Weighted	No	No	Yes	Yes	Yes
Shortest Path	Yes	No	Yes	Yes	Yes
S.P. Complexity	$O(V+E)$		$O(V+E)$	$O(V \lg V + E)$	$O(V^2)$
Feature	Levels	Edge Type /Exit Time /Topo Sort			

COMPLEXITY ANALYSIS I (COMPUTATION MODEL)

- L2: Random Access Machine: direct access
- L2: Pointer Machine: pointer
- L7: Comparison Model: Search/Sorting
    - Desision Tree (lower bound)
        - n leaves (possible answers), k branches (2 for decision tree: yes or no, 3 for comparison: <, >, =)
        - $\Omega(\log_k n)$

COMPLEXITY ANALYSIS II (TECHNIQUE)

- 几何级数（影响常数，不影响渐进复杂度，但 $\sum_{i=1}^{n} i$ 则为 $O(n^2)$, $\sum_{i=1}^{n} \frac{1}{i}$ 也会影响渐进复杂度）
    - L4: Build_Max_Heap: 几何级数的和是收敛的，因此是常数的，看似 $O(n \log n)$ 的

问题其实是 $O(n)$ 的
    - L12: 除法因为有 $\lg d$ 次的乘法，因此其复杂度看似是乘法的复杂度乘以次数，但由于几何级数，除法的渐进复杂度和乘法相同，开根号也相同
- R8: 对于BST中的 `list(tree, l, h)` ，TA详细分析了其复杂度，首先罗列的总数 I 是固定的复杂度，那么需要确定的就是不在罗列范围内需要访问的节点个数
- L9: Amortized Analysis平摊分析
    - R9: 对于BST的 `next_larger` 的平摊复杂度是常数的，因此对于 `list(l, h)` 来说，如果先找到 `l` 对应的node，再查找所有successor直到找到 `h` ， 所有步骤的复杂度即为 $O(\log n + I)$

REDUCTION

Raw Input $\overset{Transform}{\Longrightarrow}$ Input $\Longrightarrow$ **KNOWN ALGORITHMS**
$\Longrightarrow$ Output $\overset{Interpret}{\Longrightarrow}$ Expected Output

- R15: 对于weighted graph寻找最短路径，每个edge增加weight-1个节点，然后使用BFS
- R16: 想起了6.001中介绍的State Machine，当前State是什么，可能的Process是什么，经过一个Process之后的State是什么，目标State是什么。
    - 不同state使用不同的节点来表示（多个layer的节点）
- R18: Methodology
    - where you have a graph that's 2D, and we want to compute something that Dijkstra can't compute on it's own. Or that Bellman-Ford can't compute on it's own.
    - to add additional states to the graph. And the way we do that is we make copies of the graph that we call layers.
    - figuring out what those layers are and how you connect them.
- PSet6: Dijkstra寻找最短路径，为了使数据在Priority Queue中可比较，且保留路径或节点，建立首元素可比较的tuple，或是建立自定义类，定义其比较方式

ALGORITHM READ / DESIGN

- R2: 阅读代码的方式（函式呼叫图Call Graph），那么这或许也应该是书写代码的顺序，从前到后，从主要到次要，看首要函数调用了哪些其他的函数。用缩进或是树图来表示层级。
- PS2: 通过一个实例练习了算法优化：
  - 首先使用 `profiler` 找到程序中最为耗时的函数/方法
  - 然后搜索/研究/设计更优的数据结构/算法
  - 将此函数/方法使用新的数据结构/算法进行优化
    - （优化此函数的同时，注意是否影响其他函数的复杂度）
- R11: Principles of Algorithm Design (**Recitation Note**)
  - Experiment with examples.
  - Simplify the problem.
    - Sometimes, when a problem is difficult to solve, it can be worth it to solve a related, simpler problem instead.
    - Once you develop ideas for the simpler case, you can often apply them to handle the more complex case.
  - Look for similar problems.
  - Delegate the work.
    - Recursion
  - Design according to the runtime.
- R11: 从数据结构的深入理解到算法
  - 例如 k^{th} minimum 问题，如果能够理解到最小堆（min-heap）的所有子堆都是最小堆的话，只需要维护一个去除掉前 i 个最小值的子堆的根（root）的最小堆即可。
- PSet5: 常系数对小数据集的影响
  - 对于渐近复杂度较高的算法，如果其常数系数较小，在较小输入时可能有极大的优势，因此在实践中，往往需要根据不同的输入尺度选择不同的算法。

TREE TRAVERSAL

- DFS
    - Pre-order: topological sort
    - In-order: BST sort

CODING

- PSet3: 大量类、继承、递归、AVL树的经验
- PSet4: 使用字典避免多次遍历、生成（generator）
    - 查找两文件相同元素时，建立第一个文件元素的字典，而不多次遍历
- Pset6: 自上而下的问题解决思路：先写整体问题解决的代码，需要用的的helper function作为占位符写出，然后再一点点填充内容（将复杂问题简单化，乃至可以先写大框架的pseudo code）

* COURSE *

U1. INTRODUCTION

THEME

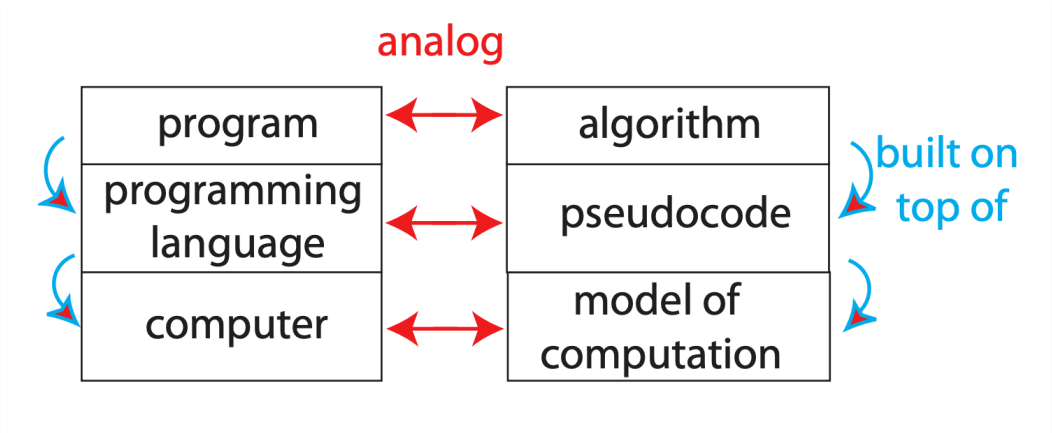
- Efficient procedures for solving problems on large inputs
- Scalability

ASYMPTOTIC COMPLEXITY

$$g(x) = (1 + \sin(x))x^{1.5} + x^{1.4}$$

$$\begin{aligned} g(x) &= \Theta(\quad) && \text{(upper and lower bound)} \\ &= \Omega\left(x^{1.4}\right) && \text{(lower bound)} \\ &= O\left(x^{1.5}\right) && \text{(upper bound)} \end{aligned}$$

COMPUTATION MODEL



- Random Access Machine
- Pointer Machine

ALGORITHM PROOF

1. If the peak problem is not empty, then **algorithm1** will always return a location.
2. If **algorithm1** returns a location, it will be a peak in the original problem.

U2. SORTING AND TREES

INSERTION SORT

Binary Insertion Sort

- for $j \leftarrow 2$ to n
  - insert key $\mathrm{A}[j]$ into the (already sorted) sub-array $\mathrm{A}[1..j-1]$. Use binary search to find the right position
- Complexity
  - $\Theta(\mathrm{n} \log \mathrm{n})$ comparisons
  - $\Theta\left(\mathrm{n}^2\right)$ swaps

MERGE SORT (DIVIDE & CONQUER)

MERGE-SoRT $A[1 \ldots n]$ —— $T(n)$

1. If $n = 1$, done (nothing to sort). —— $\Theta(n)$
2. Otherwise, recursively sort $A[1 \ldots n/2]$ and $A[n/2 + 1 \ldots n]$. —— $2T(n/2)$

3. "Merge" the two sorted sub-arrays. —— $\Theta(n)$

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1 \\ 2T(n/2) + \Theta(n) \text{ if } n > 1 \end{cases}$$

- Complexity: $\Theta(n \log n)$

PRIORITY QUEUE (HEAP)

ADT

- insert, max, extract_max, increase_key

Heap

- Implementation of a priority queue
- An array, visualized as a nearly complete binary tree
- Max Heap Property: The key of a node is $\geq$ than the keys of its children (Min Heap defined analogously)

- Operations
  - max_heapify, build_max_heap, extract_max, heap_sort
  - sift_up, sift_down

Complexity Analysis

- $\mathrm{O(n \log n)}$ via simple analysis

- Total amount of work in the for loop can be summed as:
  $n/4(1c) + n/8(2c) + n/16(3c) + \ldots + 1(\lg nc)$
- Setting $\mathrm{n/4 = 2^k}$ and simplifying we get:
  c $\mathrm{2^k \left(1/2^0 + 2/2^1 + 3/2^2 + \ldots (k+1)/2^k\right)}$
- The term is brackets is bounded by a constant!
  This means that Build_Max_Heap is $\mathrm{O(n)}$

BINARY SEARCH TREE (AVL)

二叉搜索树的特点在节点的所有左子节点的值都小于该节点的值，所有右子节点的值都大于该节点的值。因此无论是搜索还是插入都只需要树的高度的计算复杂度 $\Theta(h)$。

AVL

- balance, rotation
- sort
    - insert, in-order traversal

COMPARISON MODEL OF COMPUTATION

- input items are black boxes (ADTs)
- only support comparisons (<, >, ≤, etc.)
- time cost = # comparisons

And basically the last four lectures have all been about algorithms in this model.

LINEAR-TIME SORTING (INTEGER SORTING)

- Counting Sort
    - integers (fitting in a word) ∈ 0, 1, . . . , k − 1
    - $O(n+k)$
- Radix Sort
    - counting sort for each digit (limit number range to digit base)
    - base $b \implies \#\text{digits } d = \log_b k$
    - $\Theta\left((n+b)\log_b k\right)$

- when $b = n$ and $k \leq n^c$: $O(nc) = O(n)$

U3. HASHING

DICTIONARY PROBLEM

Abstract Data Type (ADT) — maintain a set of items, each with a key, subject to

- insert(item): add item to set
- delete(item): remove item from set
- search(key): return item with key if it exists

We assume items have distinct keys (or that inserting new one clobbers old).

Goal: $O(1)$ time per operation.

TWO PROBLEM: PREHASH / HASH

1. keys must be nonnegative integers (or using two arrays, integers)
2. large key range $\Longrightarrow$ large space - e.g. one key of 2^{256} is bad news.

1. "Prehash" keys to integers

2. Hashing $\rightarrow$ Collision

- Reduce universe $\mathcal{U}$ of all keys (say, integers) down to reasonable size m for table
- idea: $m \approx n = \#$ keys stored in dictionary
- hash function $\mathrm{h} : \mathcal{U} \rightarrow \{0, 1, \ldots, m-1\}$
- two keys $k_i, k_j \in K$ collide if $h\left(k_i\right) = h\left(k_j\right)$
    - two solutions to collision
        - 2.1.Chaining
        - 2.2.Open Addressing

2.1. Chaining

- Linked list of colliding elements in each slot of table
- Hash Functions
    - Division Method: $h(k) = k \bmod m$
    - Multiplication Method: $h(k) = [(a \cdot k) \bmod 2^w] \gg (w - r)$
    - Universal Hashing: $h(k) = [(ak + b) \bmod p] \bmod m$
- Simple Uniform Hashing Assumption
    - Each key is equally likely to be hashed to any slot of table, independent of where other keys are hashed.
    - Load Factor α:

$$\begin{aligned} \text{let } n &= \#\text{ keys stored in table} \\ m &= \#\text{ slots in table} \\ \text{load factor } \alpha = n/m &= \text{ expected } \#\text{ keys per slot} \end{aligned}$$

2.2. Open Addressing

- one item per slot $\Longrightarrow m \geq n$
    - hash function specifies order of slots to probe (try) for a key (for insert/search/delete), not just one slot
- Hash Functions (Probing Strategies)

    - Linear Probing: $h(k,i) = (h'(k) + i) \bmod m$
    - Double Hashing: $h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod m$
- Uniform Hashing Assumption
    - Each key is equally likely to have any one of the m! permutations as its probe sequence

      Double Hashing can come close
    - Suppose we have used open addressing to insert n items into table of size m. Under the uniform hashing assumption the next operation has expected cost of $\leq \frac{1}{1-\alpha}$, where $\alpha = n/m (< 1)$.
      Example: $\alpha = 90\% \implies 10$ expected probes

Open Addressing vs. Chaining

Open Addressing: better cache performance (better memory usage, no pointers needed)

Chaining: less sensitive to hash functions (OA requires extra care to avoid clustering) and the load factor α (OA degrades past 70% or so and in any event cannot support values larger than 1)

THIRD PROBLEM: SIZE

Table Doubling

- $m* = 2?m = \Theta(n)$ still $(r+ = 1)$

  $\implies$ rebuild at insertion 2^i

  $\implies n$ inserts cost $\Theta(1 + 2 + 4 + 8 + \cdots + n)$ where n is really the next power of 2 $= \Theta(n)$

  Table Doubling

- a few inserts cost linear time, but $\Theta(1)$ "on average".

Delete

- when n decreases to $m/4$, shrink to half the size $\implies O(1)$ amortized cost

Resizable Arrays:

- same trick solves Python "list" (array)
- $\implies$ list.append and list.pop in $O(1)$ amortized

STRING MATCHING

Given two strings s and t, does s occur as a substring of t? (and if so, where and how many times?)

Karp-Rabin Algorithm

- Compare $h(s) == h(t[i : i + \text{len(s)}])$

Rolling Hash

Maintain string x subject to

- r(): reasonable hash function h(x) on string x
- r.append(c): add letter c to end of string x
- r.skip(c): remove front letter from string x, assuming it is c

U4. NUMERICS

NEWTON'S METHOD

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Square Roots

$$f(x) = x^2 - a$$

$$\chi_{i+1} = \chi_i - \frac{\left(\chi_i^2 - a\right)}{2\chi_i} = \frac{\chi_i + \frac{a}{\chi_i}}{2}$$

High Precision Computation

$\sqrt{2}$ to d -digit precision: $\underbrace{1.414213562373}_{\text{d digits}}\cdots$ Want integer $\left\lfloor 10^d\sqrt{2}\right\rfloor = \left\lfloor\sqrt{2\cdot 10^{2d}}\right\rfloor -$ integral part of square root Can still use Newton's Method.

KARATSUBA'S METHOD

Multiplying two n -digit numbers $(\text{radix}\, r = 2, 10)$:
$0 \le x, y < r^n$

Let

$$\begin{aligned} x &= x_1 \cdot r^{n/2} + x_0 \quad x_1 = \text{ high half} \\ y &= y_1 \cdot r^{n/2} + y_0 \quad x_0 = \text{ low half} \\ 0 &\le x_0, x_1 < r^{n/2} \\ 0 &\le y_0, y_1 < r^{n/2} \\ z = x \cdot y &= x_1 y_1 \cdot r^n + \left(x_0 \cdot y_1 + x_1 \cdot y_0\right) r^{n/2} + x_0 \cdot y_0 \end{aligned}$$

4 multiplications of half-sized #'s $\Longrightarrow$ **quadratic algorithm** $\Theta\left(n^2\right)$ **time**

Let

$$\begin{aligned} z_0 &= x_0 \cdot y_0 \\ z_2 &= x_1 \cdot y_1 \\ z_1 &= (x_0 + x_1) \cdot (y_0 + y_1) - z_0 - z_2 \\ &= x_0 y_1 + x_1 y_0 \\ z &= z_2 \cdot r^n + z_1 \cdot r^{n/2} + z_0 \end{aligned}$$

There are three multiplies in the above calculations.

$$\begin{aligned} T(n) &= \text{ time to multiply two } n\text{-digitH's} \\ &= 3T(n/2) + \theta(n) \\ &= \theta\left(n^{\log_2 3}\right) = \theta\left(n^{1.5849625\cdots}\right) \end{aligned}$$

HIGH PRECISION DIVISION

We want high precision rep of $\frac{a}{b}$

- Compute high-precision rep of $\frac{1}{b}$ first
- High-precision rep of $\frac{1}{b}$ means $\left\lfloor \frac{R}{b} \right\rfloor$ where R is large value s.t. it is easy to divide by R , Ex: $R = 2^k$ for binary representations
- Newton's Method:

$$f(x) = \frac{1}{x} - \frac{b}{R}$$

Complexity

$$c \cdot 1^\alpha + c \cdot 2^\alpha + c \cdot 4^\alpha + \cdots + c \cdot \left(\frac{d}{4}\right)^\alpha + c \cdot \left(\frac{d}{2}\right)^\alpha + c \cdot d^\alpha < 2c \cdot d^\alpha$$

(geometric series, same for square roots)

INITIAL GUESS

For Roots

1. compute number of digits
2. Binary search for the first digit

U5. GRAPHS

GRAPH REPRESENTATIONS (DATA STRUCTURE)

- Adjacency lists: Array Adj of $|V|$ linked lists
    - multiple graphs on same vertices
- Implicit Graphs: $\mathrm{Adj}(u)$ is a function
    - requires "Zero" Space
- Object-oriented Variations
- Adjacency matrix
    - constant time to find if there is an edge between two vertices

BREADTH-FIRST-SEARCH ALGORITHM (BFS)

- $O(V + E)$ ("LINEAR TIME")

DEPTH-FIRST SEARCH (DFS)

- $O(V + E)$ ("LINEAR TIME")

Edge Classification

- to compute this classification (back or not), mark nodes for duration they are "on the stack"
- only tree and back edges in undirected graph

Cycle Detection

- Graph G has a cycle $\Leftrightarrow$ DFS has a back edge

Topological Sort on Directed Acylic Graph

- Reverse of DFS finishing times (time at which DFS-Visit(v) finishes)

U6. SHORTEST PATH

Negative-weight edges

- **negative weight cycles** $\Longrightarrow$ may make certain shortest paths undefined!

  $\Longrightarrow$ If negative weight edges are present, s.p. algorithm should find negative weight cycles (e.g., Bellman Ford)

DAG

can have negative values

1. Topologically sort the DAG. Path from u to v implies that u is before v in the linear ordering.
2. One pass over vertices in topologically sorted order relaxing each edge that leaves each vertex.

$\Theta(V + E)$ time

DIJKSTRA'S ALGORITHM

can have cycles but can't have negative values

```
Dijkstra (G, W, s) //uses priority queue Q
  Initialize (G, s)
  S ← None
  Q ← V[G] //Insert into Q
  while Q != None
    do u ← EXTRACT-MIN(Q) //deletes u from Q
    S = S U {u}
    for each vertex v in Adj[u]
      do RELAX (u, v, w) // ← this is an implicit DECREASE KEY operation
```

- Strategy: Dijkstra is a greedy algorithm: choose closest vertex in $V - S$ to add to set S.
- Correctness: We know relaxation is safe. The key observation is that each time a vertex u is added to set S, we have $d[u] = \delta(s, u)$.

Implementation of Q (Priority Queue)

- Array: $\Theta(V \cdot V + E \cdot 1) = \Theta\left(V^2 + E\right) = \Theta\left(V^2\right)$
  (where $E = O(V^2)$)
- Binary Min-Heap: $\Theta(V \lg V + E \lg V)$
- Fibonacci Heap: $\Theta(V \lg V + E)$

BELLMAN-FORD

```
Bellman-Ford(G,W,s)
  Initialize ()
  for i = 1 to |V| - 1
    for each edge (u, v) ∈ E:
      Relax(u, v)
  for each edge (u, v) ∈ E
    do if d[v] > d[u] + w(u, v)
      then report a negative-weight cycle exists
```

SPEEDING UP

Bi-Directional Search

- Subtlety: After search terminates, find node x with minimum value of df (x) + db(x). x may not be the vertex w that caused termination as in example to the left!

Goal-Directed Search or A* (AI)

U7. DYNAMIC PROGRAMMING

Technique

- Memoization
  - compute sub-problem once
- Guess
- Bottom-up (DAG)
  - practically faster: no recursion
- Parent Pointer
  - when you have the min weight, reconstruct the route

5 EASY STEPS TO DYNAMIC PROGRAMMING

1. define subproblems — count # subproblems
2. guess (part of solution) — count # choices
3. relate subproblem solutions — compute time/subproblem
4. recurse + memoize — time = time/subproblem · # subproblems
   OR build DP table bottom-up
   check subproblems acyclic/topological order
5. solve original problem: = a subproblem
   OR by combining subproblem solutions $\Longrightarrow$ extra time

DEFINING SUBPROBLEMS

* problems from L20 (text justification, Blackjack) are on sequences (words, cards)
* useful problems for strings/sequences x:

- $\Theta(n)$
  - suffixes $x[i:]$
  - prefixes $x[:i]$
- $\Theta(n^2)$
  - substrings $x[i:j]$
    - examples: Parenthesization, Edit Distance...

2 KINDS OF GUESSING

1. In (3), guess which other subproblems to use (used by every DP except Fibonacci)
2. In (1), create more subproblems to guess/remember more structure of solution used by knapsack DP
   - effectively report many solutions to subproblem.
   - lets parent subproblem know features of solution.
   - examples: Fingering...