Introduction to Algorithms MIT 6.006

By:

Dustin Smith

Contents

1 Lecture: Algorithmic Thinking, Peak Finding

Lecture 1 from the MIT's Intro to Algorithms.

Peak finding runs on in a one-dimensional array. Let the array be length of 9 indexed from 1 to 9.

a	b	c	d	e	f	g	h	i

Table 1.1: 1D array peak finding.

Definition:

A position, b, is a peak if and only if position $b \geqslant a$ and $b \geqslant c$.

Find a peak if it exists. Straightforward algorithm going from left to right. In the worst case, the complexity would be $\theta(n)$.

Definition:

Divide and Conquer (Binary Search); look at $n/2$ position, if $a[n/2] < a[n/2-1]$, then only look at the left half indices from $1, \ldots, n/2-1$ to look for a peak and vice versa. Otherwise, $n/2$ position is a peak.

What is the complexity of this divide and conquer algorithm?

$$T(n) = T(n/2) + \theta(1)$$

where $T(n)$ is the work done by the algorithm of size n. Let's list out some of $T(n/2)$.

$$\begin{aligned} n/2, n/4, n/8, n/16, \ldots \rightarrow \quad n/2^k &\leqslant 1 \\ n &\leqslant 2^k \\ \log_2 n &\leqslant k \end{aligned}$$

Therefore, the time complexity $T(n) = \theta(\log_2 n) + \theta(1) \rightarrow \theta(\log_2 n)$. Now, let's consider the 2D-peak.

	c			
b	a	d		
	e			

Table 1.2: 2D matrix peak finding.

Definition:

a is a 2D-peak if and only if $a \geqslant b$, $a \geqslant d$, $a \geqslant c$, and $a \geqslant e$.

For this case, we want to use the Greedy Ascent algorithm. With Greedy Ascent, we must make a choice where to start and where to go first: left or right or up or down. In the worst case, we have $\theta(n \cdot m)$ complexity, and when $n = m$, $\theta(n^2)$. We can apply divide and conquer for a 2D peak as follows:

1. Pick a column j to run binary search on the rows, and at row (i, j), we have a 1D peak.
2. Next check $(i \pm 1, j)$ against (i, j).

3. If $(i, j) \geqslant (i \pm 1, j)$, then we are done we have a 2D peak.
4. Otherwise, move left/right to find the new peak and then check $(i + 1, j \pm 1)$ and so on.

When we have a single column, find the global maximum and done. Overall, we have

$$\begin{aligned}
T(n, m) &= T(n, m/2) + \theta_{\text{global max}}(n) \\
T(n, 1) &= \theta(n) && \text{Base case} \\
T(n, m) &= \underbrace{\theta(m) \cdots \theta(m)}_{\text{n times}} \\
&= m^n / 2^k && \leqslant 1 \\
&= m^n && \leqslant 2^k \\
&= n \log_2 m && \leqslant 1 \\
&= \theta(n \log_2 m)
\end{aligned}$$

where this follows from the 1D time complexity.

2 Lecture: Models of Computation, Document Distance

Lecture 2 from the MIT's Intro to Algorithms.

What is an algorithm? It is a way to define computational procedure for solving some problem. An algorithm takes an input and returns some output.

1. Random Access Machine (RAM) in constant time can load, do computations, and store in constant time by memory address.

In Python, sort is $\theta(n \log n)$ and list length is constant time since Python stores the length pre-computed. Dictionary look up in Python takes constant time, $\theta(1)$. List append is constant time as well.

Definition:

Document distance problem: given two documents D_1 and D_2, I want to compute the distance between.

A document is a sequence of words and word is a string of characters. How can we do document distance? In this example, we can look at shared words. Consider two documents, $D_1 =$ "the cat" and $D_2 =$ "the dog". What are some ways we could measure distance?

$$\begin{aligned} d'(D_1, D_2) &= D_1 \cdot D_2 \\ &= \sum_w D_1[w] \cdot D_2[w] \\ d''(D_1, D_2) &= \frac{D_1 \cdot D_2}{|D_1||D_2|} \\ d(D_1, D_2) &= \arccos(d'') \end{aligned}$$

Algorithm

1. Split document into words
2. Compute word frequencies
3. Compute dot product

3 Lecture: Binary Sort, Insert Sort, Merge Sort

Lecture 3 from the MIT's Intro to Algorithms.

Why sorting? Easy to search like phone books, mp4 organizations, etc. Problems can become easy once items are sorted, example, finding a median.

```
def binary_search_recursive(array, element, start, end):
    if start > end:
        return -1

    mid = (start + end) // 2
    if element == array[mid]:
        return mid

    if element < array[mid]:
        return binary_search_recursive(array, element, start, mid - 1)
    else:
        return binary_search_recursive(array, element, mid + 1, end)

def binary_search_iterative(array, element):
    mid = 0
    start = 0
    end = len(array)

    while start <= end:
        mid = (start + end) // 2

        if element == array[mid]:
            return mid

        if element < array[mid]:
            end = mid - 1
        else:
            start = mid + 1
    return -1

def insertion_sort(array):
    # We start from 1 since the first element is trivially sorted
    for index in range(1, len(array)):
        currentValue = array[index]
        currentPosition = index

        while (currentPosition > 0 and
            array[currentPosition - 1] > currentValue):
            array[currentPosition] = array[currentPosition -1]
            currentPosition = currentPosition - 1

        array[currentPosition] = currentValue
```

Data compression benefits from sorting. Insertion sort, works by doing pair-wise swaps. What is the time complexity? We have $\theta(n)$ steps but each step could do $\theta(n)$ work (swaps and compares) so $\theta(n^2)$.

Let's consider Merge Sort, another divide and conquer algorithm. Split array into left and right halves. Then we merge and sort array. However, with merge sort, we need $\theta(n)$ auxiliary space whereas with insertion sort, we need $\theta(1)$ auxiliary space.

```
 def merge_sort(list):
    list_length = len(list)

    if list_length == 1:
        return list

    mid_point = list_length // 2

    left_partition = merge_sort(list[:mid_point])
    right_partition = merge_sort(list[mid_point:])

    return merge(left_partition, right_partition)

def merge(left, right):
    output = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            output.append(left[i])
            i += 1
        else:
            output.append(right[j])
            j += 1

    output.extend(left[i:])
    output.extend(right[j:])

    return output
```

4 Lecture: Heaps and Heap Sort

Lecture 4 from the MIT's Intro to Algorithms.

We will use heap as an example of an implementation of a priority queue.

Definition:

Priority Queue implements a set S of elements, each of the elements is associated with a key.

A heap is classified as an Abstract Data Type (ADT). We will specify the set of operations we would like to perform on a priority queue?

* Insert (S, x): insert x into a set of S
* max(S): return the element of S with the largest key
* extract-max(S): and remove from S
* increase priority key (S, x, k): increase the value of x's key to k.

A heap is an implementation of a priority queue. An array visualized as a nearly complete binary tree. With a

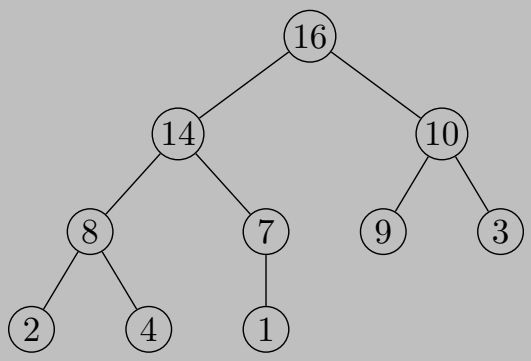

Figure 4.1: Binary Tree: follows the max heap property.

heap as a tree, the root element is the first element with $i = 1$, parent$(i) = i/2$, left$(i) = 2$, and right$(i) = 2i + 1$. The max heap property says that the key of a node is greater than or equal to the keys of its children. For min heap, replace $\geqslant$ with $\leqslant$. How do we build a max heap out of an initially unsorted array? Heap operations:

* build_max_heap: produces a max heap from an arbitrary array.
* max_heapify: correct a single violation of the heap property in a sub-tree's root.

For max_heapify(A, i), we assume that the tree's rooted at left(i) and right(i) are max heaps. For figure 4.2, we call max_heapify$(A, 2)$. What does this code do?

* Exchange $A[2]$ with $A[4]$
* Call max_heapify$A[4]$
* Exchange $A[9]$ with $A[4]$

The complexity of max_heapify would be $\theta(\log_2 n)$. How would we use max_heapify to create build_max_heap? To convert $A[a, \ldots, n]$ into a max heap, What is the complexity of build? $O(n \log_2 n)$, can we do better than this (simple analysis)? Through careful analysis, we have $\theta(n)$ complexity. Observe that max_heapify takes

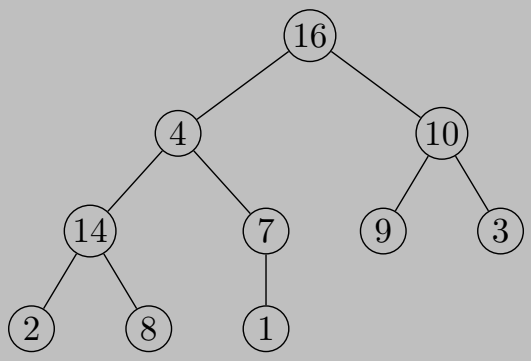

Figure 4.2: Consider this binary representation for a heap. We need to max_heapify this tree.

constant time $O(1)$ for nodes that are one level above leaves, and in general, $O(\ell)$ time for nodes that are ℓ levels above the leaves. We have $n/4$ nodes with level 1, $n/8$ with level 2, ... and 1 node level $\log_2 n$. The total amount of work in the for loop can be summed as

$$\frac{n}{4} \cdot c + \frac{n}{8} \cdot 2c + \frac{n}{16} \cdot 3c + \cdots + 1 \cdot \log_2 n \cdot c$$

Set $n/4 = 2^k$ then we have

$$c \cdot 2^k \left(\frac{1}{2^0} + \frac{2}{2^1} + \frac{3}{2^2} + \cdots + \frac{k+1}{2^k} \right) = c \cdot 2^k \cdot \sum_{i=0}^{k} \frac{i+1}{2^i} \leqslant M$$

where M is a constant. Thus, we have $\theta(c \cdot n/4 \cdot M) \to \theta(n)$ complexity.

1. Build max heap from unsorted array; $\theta(n)$
2. Find max element $A[1]$
3. Swap elements $A[n]$ with $A[1]$; now the max element is at the end of the array
4. Discard node n from the heap simply by decrementing the heap size.
5. New root after the swap may violate the max heap property but the children are max heaps
6. Run max_heapify

The complexity is $\theta(n \log_2 n)$.

```python
# max_heapify pseudo python
def max_heapify(A, idx):
    l = A.left(i)
    r = A.right(i)
    n = len(A)

    if l <= n and A[l] > A[i]:
        largest = l
    else:
        largest = i
    if r <= n and A[r] > A[largest]:
        largest = r
    if largest != i:
        A[i] = A[largest]
        A[largest] = A[i]
        max_heapify(A, largest)
```

```python
# build_max_heap pseudo python
def build_max_heap(A):
    for i in range(n // 2, 0, -1):
        max_heapify(A, i)
```

5 Lecture: Binary Search Trees, BST Sort

Lecture 5 from the MIT's Intro to Algorithms.

Consider the runway reservation system problem. Let's assume this airport has a single runway. We are working on reservations for future landings. We need to reserve requests for landings time, t. Add t to the set R of landing times if no other landings are scheduled within k minutes. After the plane lands, remove from R. We will do all these operations in $\theta(\log_2 n)$ where n is the size of the set. For this example, let $k = 3$. That is, $|t_i - 3| \geqslant 3$ to insert a new t_i in set R. Additionally, we cannot insert times in the past.

1. The set cannot be unordered array; we cannot use binary search.
2. With a sorted array, we can find the insertion point in order $\theta(\log_2 n)$, $\theta(1)$ to check k against $t_{i\pm1}$, but it will take order $\theta(n)$ to insert and shift.
3. With a sorted linked list, we could insert in $\theta(1)$ but we couldn't use binary search.
4. With heaps (min or max), an element that is $\leqslant k$ of $\geqslant k$ from t then $\theta(n)$ time.
5. Dictionaries would be $\theta(n)$ as well.

If we could we do fast insertion into a sorted linked list, we would be in business.

Binary search trees Each node x has a key for x. Unlike a heap, the BST has pointers that are parent, left,

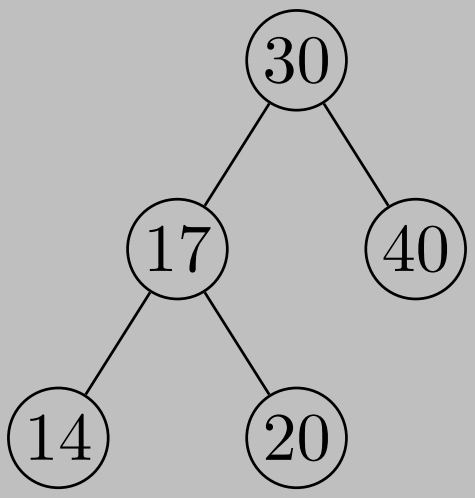

Figure 5.1: Binary Search Tree

and right child of x. For all nodes x, if y is in the left subtree of x, then $\text{key}(y) \leqslant \text{key}(x)$. If y is in the right subtree, then $\text{key}(y) \geqslant \text{key}(x)$. From a null BST, we have

1. Insert 49
2. Insert 79 but since $79 > 49$, we attach to the right child of 49
3. Insert 46 and since $46 < 49$, we attach to the left child of 49
4. Insert 41 then $41 < 49$ so go left and $41 < 46$ so attach to the left child of 46
5. Try to insert 42. $42 < 49$ so go left, $42 < 46$ so go left, and $42 > 41$ but not k more than 3 insertion fails.

If h is the height of the tree, then insertion with or without the check, is done in order $\theta(h)$ time. In BST, we can find the min or the max by going to the left or right until leaf, respectively. These would be in $\theta(h$. New requirement: rank t how many planes are scheduled to land at times $\leqslant t$? How can we do this without changing the complexity? Let's augment the BST structure. We can add another value to the node key. When we do insert and delete, we modify the subtree size numbers. A single node would be 1, a node with

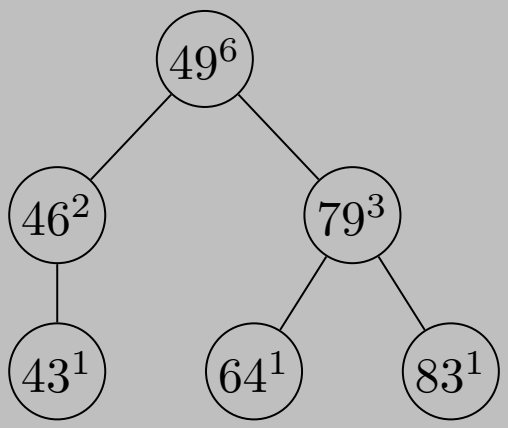

Figure 5.2: With an augmented BST, we would append number of subtrees to each node.

2 children below would be 3, and root would be the number of nodes. That is, just the number of nodes underneath parent plus itself. What lands before t in order $\theta(h)$ time?

1. Walk down the tree to find the desired time
2. Add in the nodes that are smaller
3. Add in the subtree sizes to the left

Let $t = 79$. What is $\leqslant t$?

1. 49 is clearly less than t so add 1
2. Move to the right but first add the sub tree sizes to the left; add 2
3. 79 is equal to so add 1
4. Look left and 1 from the subtree to left node 64

We have 5. See lecture 6 for how to get $\theta(\log_2 n)$ complexity.

6 Lecture: AVL Trees, AVL Sort

Lecture 6 from the MIT's Intro to Algorithms.

Balanced BST (picking up from lecture 5). We use in order traversal to get the sorted order of a BST. A perfectly balanced BST would have height roughly $\theta(\log_2 n)$.

Definition:

The height of a BST is the length of the longest path from the root the the leaf. We call a tree balanced if the height is order $\theta(\log_2 n)$.

Definition:

The height of a node in a tree is the longest path from that node to the leaf.

All leafs have height 0. The max of the height of the left child with the height of the right child plus one is the height of the parent.

AVL trees require height of left and right children of *every* node to differ by at most ± 1. Let h_l and h_r be the heights of the left and right subtrees, respectively. Then we want $|h_l - h_r| \leqslant 1$. Less than or equal to zero is almost impossible to maintain but 1 is more reasonable. We claims that AVL trees are balanced; worst case is when right (or left) subtree has height 1 more than the left for every node. Let $N_h =$ min number of nodes in the an AVL tree of height h. The base case is $N_{\theta(1)} = \theta(1)$. We have that $N_h = 1 + N_{h-1} + N_{h-2}$. Let $\varphi \approx 1.618033$ be the golden ratio. Then

$$N_h > F_h = \begin{cases} \left\lfloor \frac{\varphi^h}{\sqrt{5}} \right\rfloor, & \text{if the decimal portion is } \leqslant 0.5 \\ \left\lceil \frac{\varphi^h}{\sqrt{5}} \right\rceil, & \text{otherwise} \end{cases}$$

where F_h is the h-th Fibonacci number. Now we have that

$$\begin{aligned} \varphi^h/\sqrt{5} &< n \\ h \cdot \log_\varphi 1/\sqrt{5} &< \log_\varphi n \\ h \cdot c &< 1.44 \log_2 n \end{aligned}$$

where $c = \log_\varphi 1/\sqrt{5}$ is a small constant. Another approach would be to

$$\begin{aligned} N_h &= 1 + N_{h-1} + N_{h-2} \\ &> 1 + 2 \cdot N_{h-2} \\ &> 2 \cdot N_{h-2} \\ &= \theta(2^{h/2}) \end{aligned}$$

Therefore, $h < 2 \cdot \log_2 n$. AVL insert

1. simple BST insert
2. fix AVL property with BST rotation from the changed node up

What can we do with AVL sort?

* Insert n items $\theta(n \log_2 n)$
* in order traversal take $\theta(n)$

With abstract data types, we should support insert, delete, find min or max, and successor or predecessor.

7 Lecture: Counting Sort, Radix Sort, Lower Bounds for Sorting

Lecture 7 from the MIT's Intro to Algorithms.

When is linear time sorting both possible and not possible? Let's start with the comparison model.

* All input items are black boxes (ADTs)
* Only operations allowed are comparisons $(<, \leqslant, >, \geqslant)$
* Time cost will just be the number of comparisons

If we know that our algorithms are only comparing items, we can draw all the possible things it can do.

Definition:

Decision tree: any comparison algorithm can be viewed as a tree of possible comparisons and their outcomes, and the resulting answer for any particular value of n.

Decision Tree	Algorithm
internal node	binary decision
leaf	found answer
root to leaf path	algorithm execution
length of path	running time of that execution
height of the tree	worst-case running time

Table 7.1: Decision tree comparison to algorithm.

What would be the lower bound for searching? n preprocessed items finding a given item among them in the comparison model requires $\Omega(\log_2 n)$ in the worst-case. At each step, we only have a no or a yes (binary), and the tree must have all possible answers so we have at least n leaves. for the decision tree. Therefore, the height has to be at least $\log_2 n$.

What would be the lower bound for sorting? Again, we have the decision tree is binary but how many leaves does it have? The leaf will have the sorted order $A[5] \leqslant A[7] \leqslant A[0] \leqslant \cdots$. We have that the number of leaves is greater than or equal to the possible answers which would be $n!$. The height is at least

$$\begin{aligned}
\log_2 n! &= \log_2 \left[n \cdot (n-1) \cdot (n-2) \cdots 1\right] \\
&= \log_2 n + \log_2 (n-1) + \cdots + \log_2 2 + \log_2 1 \\
&\leqslant \log_2 n + \log_2 n + \cdots + \log_2 n \\
&= n \cdot \log_2 n \\
&= \Omega(n \cdot \log_2 n)
\end{aligned}$$

Another way to show this is to note that $n! \leqslant n^n$.

Linear-time sorting (integer sorting).

* Assume n keys sorting are non-negative integers in some range and each fits in a word.
* Can do a lot more than comparisons.
* For $k = n^{O(1)}$ not too large, we can sort in linear time.

Currently, the best algorithm to date performs in $O\left(n\sqrt{\log_2 \log_2 n}\right)$, not covered here. We can achieve linear time with Counting Sort when k isn't too large.

```
# pseudocode
L = array of k empty lists  # O(k)
for j in range(n):
  # key is an integer between 1 and k - 1
  L[key(A[j])].append(A[j])  # O(1)

output = []
for i in range(k):
  output.extend(L[i])  # O(L[i])

# O(n + k)
```

Radix sort uses counting sort as a sub routine.

* Imagine each integer as base b; number of digits $d = \log_b k$.
* Sort the integers by the least significant digit, the next least, ..., the most
* Sort by digit using counting sort; $O(n + b)$

The total time would be $O((n + b) \cdot d) = O((n + b) \cdot \log_b k)$. How do we minimize this? We need b to be large but not bigger than n. Therefore, we have $O(n \cdot \log_n k)$. If $k \leqslant n^c$, then $O(n \cdot c)$.

8 Lecture: Hashing with Chaining

Lecture 8 from the MIT's Intro to Algorithms.

Definition:

Dictionary is an abstract data type (ADT) which maintains a set of items each with a key. The following operations are defined: insert item, delete item, and search for a key and return the item with a given key or return a KeyError.

With dictionaries, keys are unique so inserting an existing key will overwrite the current item value. All of the dictionary operations can be done in $\theta(\log_2 n)$ time via an AVL tree. How can we search faster? We can get down to constant time with high probability.

The simple approach to a dictionary is the direct-access table. We store items in an array which are indexed by key, integers. Why isn't this optimal?

1. Keys may not be integers
2. Uses a lot of memory if we have a lot of keys

What if our keys aren't integers? We can use prehashing. Prehashing maps keys to non-negative integers. In theory, keys are finite and discrete (string of bits). In Python, $\mathtt{hash}(x)$ is the prehash of x. There are some issues here though such as $\mathtt{hash}(\text{"}\backslash\varnothing B\text{"}) = \mathtt{hash}(\text{"}\backslash\varnothing\backslash\varnothing C\text{"}) = 64$. Ideally, $\mathtt{hash}(x) = \mathtt{hash}(y)$ if and only if $x = y$ but this isn't always true in Python. How do we reduce space? Hashing. Reduce the universe of all possible keys down to a reasonable set of size m where hashing function is define as

$$h : \mathfrak{U} \to \{0, 1, \ldots, m-1\}.$$

We would like $m = \theta(n)$ where n is the number of keys in the dictionary. When $\mathtt{hash}(k_i) = \mathtt{hash}(k_j)$ but $k_i \neq k_j$, we have a collision. We can deal with this through chaining. We store is as a linked list of colliding elements. At the collision, we store a linked list with pointers to each item pointing to the next. This is hashing with chaining. In the worst case, we have $\theta(n)$.

Assume we have simple uniform hashing (unrealistic) but convenient for analysis. Each key is equally likely to be hashed to any slot in the table independently of where other keys mapped. Under this assumption, we can analyze hashing with chaining. What is the expected length of a chain for n keys with m slots? The chance of a key going to a spot is n/m. Let $\alpha = n/m$ which is the load factor of the table. As long as $m = \theta(n)$, this will be constant, $\theta(1)$. In general, the running time is $\theta(1 + \alpha)$.

Hash functions

1. Division method: $h(k) = k \mod m$ if m is prime and not close to powers of 2 or 10, this works pretty well.
2. Multiplication method: $h(k) = \left\lfloor (a \cdot k) \mod 2^w \right\rfloor \gg (w - r)$ we are assuming we are in a w bit machine where we want the left most bit of the right most w bits, r. Caveat, a should be odd and not a power of 2.
3. Universal hashing: $h(k) = \left\lfloor (a \cdot k + b) \mod p \right\rfloor \mod m$ where p is prime, a is random and $k, b \in \{0, \ldots, p-1\}$ such that $p > |\mathfrak{U}|$.

For worst case keys $k_1 \neq k_2$, the probability of $h(k_1) = h_2(k_2)$ is $1/m$.