Lecture 1

Fast Exponentiation

$$a^b \mod c$$

$$f(a,b,c) = \begin{cases} 1 \mod c & \text{if } b \equiv 0 \\ (f(a,b/2,c))^2 & \text{if } b > 0, b \equiv 0 \mod 2 \\ a * f(a,b-1,c) & \text{else} \end{cases}$$

$$T(n) = \Theta(\log b \cdot M(c))$$

Stock Problem You are given an array $A[0..n-1]$ of stock prices for n consecutive days, and want to pick two days i_0 and j_0, with $0 \leq i_0 \leq j_0 < n$ such if you buy a share of stock on day i_0 and sell it on day j_0 you have maximum gain. That is, you want to maximize $A[j_0] - A[i_0]$.

```
def lin(A):
    """ return best gain, computed by simple linear-time alg
        running time is Theta(n)
    """
    n = len(A)
    # opt will equal the optimum gain max_{i0} max_{j0>=i0} A[j0]-A[i0]; buy will equal minimum A[j] seen so far
    opt, buy = 0 = A[0]
    for j in range(1, n):
        buy = min(buy, A[j])
        opt = max(opt, A[j]-buy)
    return opt
```

Asymptotics

Strategies

1) Take the limit of the ratio of the two functions being compared. If it tends to infinity, the numerator is larger. If it tends to 0, the denominator is larger. For example, suppose we want to compare 2^n to 3^n. Then we see that: $\lim_{n\to\infty} = \frac{2^n}{3^n} = (\frac{2}{3})^n \to 0$

2) Another useful technique is to transform the two functions by a strictly increasing function, like $\log n$ or $\sqrt{n}$, and compare the behavior of the functions then. For instance, it may be tricky to see how $2^{(\log_2 n)^2}$ compares to $n^{log_2 n}$ but if we take $\log_2$ of these, we see that they are asymptotically the same. Be careful to make sure you understand how constants transform under these functions. For instance, if you apply $\log_2$ to n^2 and n^3, the results are within a constant factor of each other. Still, after applying log, functions are asymptotically the same only if they are within a constant **additive** factor of each other.

3) Finally, you can use L'hopital's rule to determine asymptotic behavior. The rule states that: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{f'(n)}{g'(n)}$

Note that we can repeatedly take the derivative until a clear trend is found. For an example, suppose we are trying to compare $\log n$ to $n^{0.01}$. If we take the ratio of their derivatives, we obtain: $\lim_{n\to\infty} \frac{\frac{1}{n}}{\frac{.01}{n \cdot 99}} = \lim_{n\to\infty} \frac{100}{n^{0.01}} \to 0$. This implies that $n^{0.01}$ is asymptotically greater.

Recurrence Relations `Runtime of original problem` = `Runtime of reduced problem` + `Time taken to reduce problem`:

$$T(n) = a \cdot T(\frac{n}{b}) + f(n)$$

- a - the # of subproblems that the original problem is divided into
- $\frac{n}{b}$ - the size of each subproblem
- $f(n)$ - how long it takes to divide into subproblems and combine the results of the subproblems

Recursion trees One way to solve recurrences is to draw a recursion tree where each node in the tree represents a subproblem and the value at each node represents the amount of work spent at each subproblem. The root node represents the original problem. In a recursion tree, every node that is not a leaf has a children, representing the number of subproblems it is splitting into. To figure out how much work is being spent at each subproblem, first find the size of the subproblem with the help of b, then substitute the size of the subproblem in the recurrence formula $T(n)$, then take the value of $f(n)$ as the amount of work spent at that subproblem. Long story short, a node with a problem size of x, the node will have a children each contributing $f(\frac{x}{b})$ amount of work.

The work at the leaves is $T(1)$, since at that point we have divided the original problem up until it can no longer be further divided. Note that this means that the work contributed by the leaves are $O(1)$.

Once we have our tree, the total runtime can be calculated by summing up the work contributed by all of the nodes. We can do this by summing up the work at each level of the tree, then summing up the levels of the tree.

- **Example 1**: **(Merge sort)** Merge sorting a list involves splitting the list in two and recursively merge sorting each half of the list. $a = 2$ since we call merge sort twice at every recursion (once on each half). $b = 2$ since each of the new subproblem has half the elements in the original list. $f(n) = O(n)$ since combining the results of the subproblems, the merge operation, is $O(n)$. The merge sort recursion tree is considered somewhat balanced. The work done at each level stays consistent (in this case, it is $O(n)$ at each level) so the total work done can be calculated by multiplying the work at each level by the number of levels, hence $O(n \log n)$ running time for merge sort. However, there are two other cases that may happen.
- **Example 2**: If the work at each level geometrically decreases as we go down the tree, then the work done at the root node (i.e. $f(n)$) will dominate the runtime.
- **Example 3**: If instead the work at each level geometrically increases as we go down the tree, then the work done at the bottom level (i.e. number of leaves $\times f(1)$ or $O($ number of leaves $)$ or $O(n \log_b a)$) will dominate the runtime.

Master Theorem **Case 1**: If $f(n) = O(n^{\log_b a - \epsilon})$ then the amount of work per level geometrically increases. $T(n) = \Theta(n^{\log_b a})$

Case 2: If $f(n) = \Theta(n^{log_b a} log^k n)$ then the amount of work per level have about the same cost (i.e. work per level does not *polynomially* increase or decrease, though work per level still may increase or decrease at some slower rate). We have to take the work of all levels into account and $T(n) = (n^{\log_b a} \log^k n \log n) = \Theta(n \log_b a \log^{k+1} n)$.

Case 3: If $f(n) = O(n^{\log_b a + \epsilon})$ then the amount of work per level geometrically decreases as we go down the tree. The work at the root level dominates and $T(n) = \Theta(f(n))$.

Example 1: $T(n) = 2T(\frac{n}{2}) + 1$, **Example 2**: $T(n) = 2T(\frac{n}{2}) + n$ (merge sort), **Example 3**: $T(n) = 3T(\frac{n}{2}) + O(n)$ (Karatsuba multiplication), **Example 4**: $T(n) = 4T(\frac{n}{2}) + n^3$ ((naive) integer multiplication)

Change of Variables For some particularly tricky recurrences, it may be necessary to combine the existing methods we've looked at with a method known as change of variables. The idea behind change of variables is to rewrite a tricky recurrence in terms of a new variable that is somehow related to the old variable, solve the new recurrence, then change back to the original variable. Consider, for example, the following recurrence containing a square root:

$$T(n) = 2T(\sqrt{n}) + (\log n)$$

None of our existing methods provide an easy way to solve this recurrence. To solve this we can perform a change of variables by defining a new variable m as $n = 2^m$. Substituting this in gives:

$$T(2^m) = 2T(\sqrt{2^m}) + \Theta(\log 2^m) = 2T(2^{\frac{m}{2}}) + \Theta(m)$$

This recursion still isn't in the standard form that we expect for Master Theorem, so we define a new recurrence, S as $S(m) = T(2^m) = T(n)$. This gives us:

$$S(m) = T(2^m) = 2T(2^{\frac{m}{2}}) + \Theta(m) = 2S(\frac{m}{2}) + \Theta(m)$$

Suddenly, we have a form we can work with using any of the previous methods we've learned. Using Master Theorem, for example, gives us $S(m) = m \log m$. Now we simply have to substitute back to make this meaningful in the context of T and n:

$$S(m) = \Theta(m \log m)$$

$$T(n) = \Theta(\log n \log \log n)$$

Sorting

Insertion Sort (in place)

```
for i in range(1,n):
  """ assume A[:i] is sorted
      goal: put A[i] in right position
  """
  while A[i] < A[i -1]: # A[-1] = -infinity
    swap(A[i], A[i-1])
    i -= 1
```

$T(n) = O(n^2)$ Worst case is reverse sorted list. E.g. $A = [n-1, ..., 1, 0]$

Divide and Conquer $T(n) =$ divide time + combine time

Merge Sort (not in place)

```
def merge_sort(A):
  n = len(A)
  if n == 1:
    return A
  L, R = merge_sort(A[:n/2]), merge_sort(A[n/2:])
  return merge(L, R)

def merge(L,R):
  l = r = 0 #counters
  output = []
  #walk along each array, taking the smallest element as we go
  while l < len(l) and r <len(R):
    if L[l] <= R[r]:
      output.append(L[l])
      l += 1
    else:
      output.append(R[r])
      r += 1
  return output
```

Complexity:

1) divide: $\Theta(n)$
2) recursion: $n_1 = n_2 = \frac{n}{2}$, $T(n_1) + T(n_2) = 2T(\frac{n}{2})$
3) merge: $\Theta(1)$ per output element, $\Theta(n)$ total

$T(n) = 2T(\frac{n}{2}) + \Theta(n) = \Theta(n \log n)$

Priority Queue

Operations `insert(S,x)`: insert `x` into set `S`

`max(S)`: return element of S with largest key

`extract_min(S)`: return element of S with largest key and remove it from S

Max Heap Property Key of a node is $\geq$ keys of its children

Heap as a tree

- **root of tree**: first element in array, corresponds to $i = 1$
- `parent(i) = floor(i/2)`: returns index of node's parent
- `left(i) = 2i`: returns index of node's left child
- `right(i) = 2i`: returns index of node's right child

Heap Operations `max_heapify`: correct a single violation of the heap property in subtree at its root

- Assume that the trees rooted at `left(i)` and `right(i)` are max-heaps
- If element `A[i]` violates the max-heap property, correct violation by "trickling" element (replace with largest child) `A[i]` down the tree, making the subtree rooted at index i a max-heap
- $T(n) = O(\log n)$

`extract_max(S)`: return element of `S` with largest key and remove it from `S`

- Swap the root with the last element of the heap
- Decrease the heap size by one
- `max_heapify` the root

$T(n) = O(\log n)$

`build_max_heap:` produce a max-heap from an unordered array

```
def build_max_heap(A):
  for i in reversed(range(1,n))
   max_heapify(A,i)
```

- $T(n) = O(n)$

`heap_sort` - just extract_max `n` times

Binary Trees

BST Property For any node `x`:

– For all nodes `y` in the `left` subtree of `x`: `key[y]` $\leq$ `key[x]`

– For all nodes `y` in the `right` subtree of `x`: `key[y]` $\geq$ `key[x]`

Note: BSTs **not** unique for a given set of keys

Operations `insert(k)`: inserts a node with key k

- Perform top-to-bottom search in the tree to identify the right place to insert the new key into the tree
- Add a node to the BST at that position

$T(n) = O($ height of tree $)$

– `find-min`: returns the node with the minimum value of the key

```
def find_min(x):
  current_node = x
  while current_node.left != None:
    current_node = current_node.left
  return current_node
```

- $T(n) = O($ height of tree $)$

`delete(x)`: deletes a node `x`, making the necessary adjustments to the BST to maintain its invariants

- **Case 1**: `x` has no children. Just delete it (i.e. change its parent node so that it doesn't point to `x`).
- **Case 2**: `x` has one child. Splice out `x` by linking `x`'s parent to `x`'s child.
- **Case 3**: `x` has two children. Splice out `x`'s successor and replace `x` with `x`'s successor.

```
def delete(node):
  #Case 1 and 2 - node has no children or 1 child
  if not node.left or not node.right:
    if node is node.parent.left:
      node.parent.left = node.left or node.right
      if node.parent.left is not None:
        node.parent.left.parent = node.parent
    else:
      node.parent.right = node.left or node.right
      if node.parent.right is not None:
        node.parent.right.parent = node.parent
      return node
  #Case 3
  else:
    successor = successor(node)
    node.key, successor.key = successor.key, node.key
    return delete(successor)
```

- $T(n) = O($ height of tree $)$ (from the successor) call

`find(k)`: returns a node with the key `k` (if it exists)

```
def search(node, k):
  if not node:
    return None
  else:
    if node.key == k:
      return node
    elif k < node.key:
      return search(node.left, k)
    else: # k > node.key
      return search(node.right, k)
```

- $T(n) = O($ height of tree $)$ `successor(x)`: finds the next node after node x

```
def successor(node):
  if not node.right:
    y = node.parent
    x = y.right
    while not y is None and x is y.parent:
      x = y #last considered node
      y = y.parent #currently considered node
    return y
    #basically, move left, but move right ASAP

  else: #look for smallest element in node's right subtree
    return find_min(node.right)
```

- $T(n) = O($ height of tree $)$

AVL Trees

Augmentation In order to have the height of the AVL Tree be $O(\log n)$, keep track of every node's height.

- Leaves have height 0
- `None` has height -1

AVL Property Inherits all BST properties

Invariant: for every node `x`, the heights of its `left` child and `right` child differ by at most 1. (update the height every time a node's subtree changes)

Note: AVL Tree **not** unique for a given set of keys

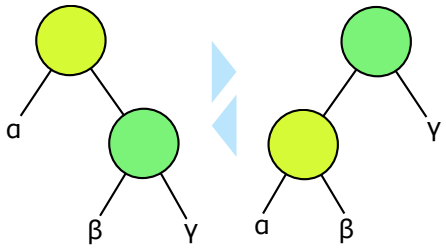

Figure 1: Rotate Left and Right

AVL Rotations

```
def AVL_rotate_left(node):
  #make sure that right subtree is non-empty
  assert node.right is not None
  old_right = node.right
  old_right_left = node.right.left

  # instead of old right node, old root takes old rights left subtree
  node.right = old_right_left

  # node's children changed - we need to recompute the height
  node.recompute_height()
  # old root becomes left node of old right (which becomes new root)
  old_right.left = node
  # old_right's children changed - we need to recompute the height
  old_right.recompute_height()

  return old_right

def AVL_rotate_right(node):
  """Rotates the tree right and returns a new root."""
  assert node.left is not None, \
  "Tree can only be rotated right if the left subtree is nonempty"

  old_left = node.left
  old_left_right = node.left.right

  node.left = old_left_right
  node.recompute_height()

  old_left.right = node
  old_left.recompute_height()
  return old_left
```

Case 1: Right-Right case

This can be fixed with a single left rotation.

```
tree.root = AVL_rotate_left(tree.root)
```

Case 2: Right-Left case

Normally, for the perfectly balanced subtrees (balance factor of 0) left-rotation decreases height of right subtree by 1 and increases the height of the right subtree by 1.

As an additional side-effect the right-left subtree (the one rooted in 6 in the example) moves from right to left subtree. If that subtree is deepest (as in our example) it results in additional decrease by 1 for the right subtree and 1 increase in the left subtree. In which case the total balance is -2 for right subtree and $+2$ for left subtree, which means our tree is unbalanced again.

Notice that this situation only happens if right-left (root 6 in example) subtree is higher than right-right subtree (root 8 in our example). Is there a way to cope with such a situation? You guessed it - we need to right-rotate the right subtree (then right-left subtree is not longer higher than right-right subtree and we land at the familiar Case 1: right right).

Let's see this in action!

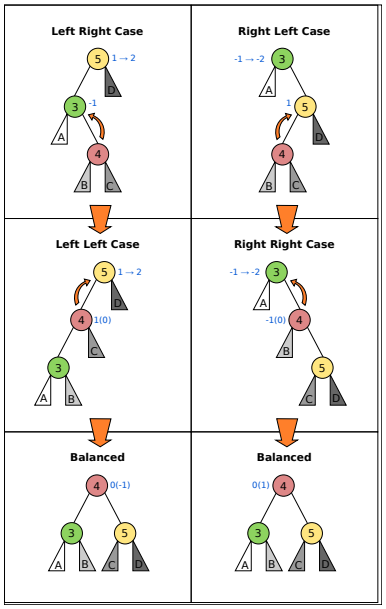

Figure 2: AVL Rotations

```
# Step 1:
tree.root.right = AVL_rotate_right(tree.root.right)
tree.root.recompute_height()

# Step 2:
tree.root = AVL_rotate_left(tree.root)
```

Cases 3 & 4: left-left and right-left

Those two cases are analogous to the cases 1 and 2, but used when the left tree is higher.

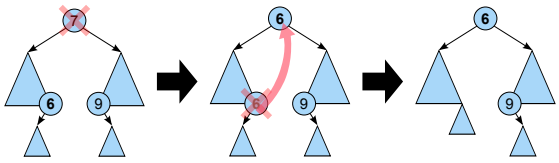

Figure 3: AVL Deletion

Linear Time Sorting

Comparison Sorting Lower Bound A decision tree can model the execution of any comparison sort: - One tree for each input size n. - A path from the root to the leaves of the tree represents a trace of comparisons that the algorithm may perform. - The running time of the algorithm = the length of the path taken. - Worst-case running time = height of tree.

Any decision tree that can sort n elements must have height $\Omega(n \lg n)$. Proof. (Hint: how many leaves are there?) - The tree must contain $\geq n!$ leaves, since there are $n!$ possible permutations of a size n array

- A height-h binary tree has $\leq 2^h$ leaves

$$
\begin{aligned}
2h &\geq n! && \\
h &\geq lg(n!) && \text{lg is monotonically increasing} \\
&\geq \lg((\frac{n}{e})^n) && \text{Stirling's formula} \\
&= n \lg n n \lg e && \\
&= \Omega(n \lg n) &&
\end{aligned}
$$

Counting Sort Assume we only have elements $0, 1, ..., (k-1)$ in the array. We know that all zeros come before all ones etc. We can therefore put all the numbers in k different buckets and later read them off.

```
def count_sort(array, k, key=lambda x: x):
  """Stable sorts array by using key to determine ordering of elements.
  Assumes all elements are in range(0, k)"""
  # initialize array
  buckets = [[] for _ in range(k)]
  # for every key store all the elements with that key
  for element in array:
    buckets[key(element)].append(element)
    output = []
  # red numbers from buckets in order
  for bucket in buckets:
    for element in bucket:
      output.append(element)
  return output
```

We have the following steps: 1) allocate space for b buckets: $O(b)$, 2)loop through all the elements in the input array and put them in buckets $O(n)$, 3)remove elements from the buckets $O(n)$

Therefore the total complexity is $O(n+b)$. Note that in the case where you're trying to sort something that ranges from 1 to 10000, but you only have 10 elements, b will dominate the runtime. This seems wasteful.

Radix Sort Imagine that you want to compare two long numbers. For example 85823421348134214 and 85823421348452456. The algorithm you would use is to compare the first digit and if it is the same then compare the next digit etc. We can say that first digit is the primary comparison criterion, second digit is the secondary sorting criterion etc. This is almost correct, but we actually need to make sure that we add extra zeros at the beginning of the number that is shorter (because sorter numbers come before longer numbers).

Radix sort uses this idea directly for sorting. It first sorts the numbers by last digit. The it stable-sorts it by the second to last digit (making second to last digit primary sorting criterion and the last digit secondary sorting criterion) and so on. At the end of that process we end up with digits sorted in exactly the order we discussed above.

```
def ith_digit(number, i):
  """Returns the i-th digit from the end.
  i=0 resuts the very last digit."""
  for _ in range(i):
    number /= 10
  return number % 10

def radix_sort_by_ith_digit(array, b, i):
  """Returns array sorted by i-th digit from the end (base b).
  The sorting procedure is stable."""
  return count_sort(array, b, key=lambda number: ith_digit(number, b, i))

def radix_sort(array, b):
  """Returns array sorted by i-th digit from the end.
  The sorting procedure is stable."""
  i = 0
  while True:
    if is_sorted(array):
      # we stop once the array is sorted the latest this can happen is when we run the number of passes equal to
      # the length of the longest number
      break
    # stable sort by i-th digit.
    array = radix_sort_by_ith_digit(array, b, i)
    i += 1
  return array
```

Let b be the base and n size of the array. Moreover let's assume that all the numbers in the array are less than or equal a. A single iteration of count sort is $O(n+b)$. How many iterations are there? At most as many as the number of digits in the longest number: $O(\log_b a)$. Therefore the total complexity of the algorithm is $O((n+b)\log_b a)$. In theory we often assume that both b and a are constants - they are after all independent of n - they won't influence the run time as n grows. That's why some theorists say that Radix Sort is $O(n)$.