6.006
Scott You

Problem Set 1 - Introduction to Algorithms

1) Functions in order of growth:

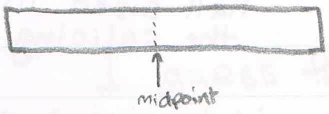
$$1/n,\ 1/5,\ 10^{100}n,\ n\log n,\ n^{100},\ 3^{\sqrt{n}},\ 3^n,\ 4^n,\ 2^{2n},\ \log(n!),\ \binom{n}{100}$$

Equivalence Classes:

$$\boxed{1/n},\ \overset{\text{constant}}{\boxed{1/5}},\ \overset{\text{linear}}{\boxed{10^{100}n}},\ \overset{\text{loglinear}}{\boxed{n\log n}},\ \overset{\text{polynomial}}{\boxed{n^{100}}},$$

$$\underbrace{\boxed{3^{\sqrt{n}}},\ \boxed{3^n},\ \boxed{4^n,\ 2^{2n}}}_{\text{exponential}},\ \boxed{\log(n!)},\ \underbrace{\boxed{\binom{n}{100}}}_{\text{factoria}}$$

2) a) Iterative version runtime:



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

Solve recurrence relation using substitution method:

1) Guess the order of the algorithm is:

$O(\lg n)$

2) Show: $T(n) \le c \cdot \lg n$, for $n \ge n_0$

i) Assume $T(n/2) \le c\lg(n/2)$

ii) $T(n) \le c\lg(n/2)+1$
$= c\lg(n) - c\lg(2)+1$
$= c\lg(n) - c+1$

Try again: Guess $T(n) \le c\lg n - b$

$$T(n) \le c\lg(n/2) + 1$$

$$T(n) \le c\lg n - c + 1$$

which for $b = 1-c$

$$T(n) \le c\lg n - b$$

$$\therefore T(n) = O(\lg n)$$

b) Recursive Version Runtime:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ \underbrace{T(n/2)}_{\text{solve subproblem of half}} + \underbrace{n/2}_{\text{cost of creating new, half-sized list via the splicing operator}} & \text{if } n > 1 \end{cases}$$

Solve Recurrence relation using substitution

1) Guess: $T(n) = O(n\lg n)$

2) Show: $T(n) \le cn\lg n$ for $n > n_0$

i) Assume: $T(n/2) \le c(n/2)\lg(n/2)$

ii) Substitute:

$$\begin{aligned} T(n) &\le c(n/2)\lg(n/2) + n/2 \\ &\le cn\lg(n/2) + n \\ &= cn\lg n - cn\lg 2 + n \\ &= cn\lg n - cn + n \\ &\boxed{\le cn\lg n} \end{aligned}$$

$$\therefore T(n) = O(n\lg n)$$

c) The extra running time of the recursive function is because the splicing operat in Python creates a <u>new</u> array with n/2 elements (which I assume has been implemented to run in O(n) time.

A better implementation would be:

def binarySearch(alist, item, first=0, last

which would create no new instances of the array, like the iterative version.

3) a) Hypothesis for the order of set.intersection

O(argmin(|s|, |t|))

I guess that the order of the function intersection would be linear on the smaller of the two sets since any intersection command would need to run --contains-- at least once for each element in the smalle list.

b)

time in μs	$\lvert s\rvert = 10^3$	$\lvert s\rvert = 10^4$	$\lvert s\rvert = 10^5$
$\lvert t\rvert = 10^3$	0.003	0.003	0.003
$\lvert t\rvert = 10^4$	0.003	0.03	0.04
$\lvert t\rvert = 10^5$	0.003	0.04	0.4
$\lvert t\rvert = 10^6$	0.004	0.06	1.4

The above values (in seconds) are the total of running timeit on s.intersection(t) 100 times for varying values of |s| and |t|

c) Based on this experimentally observed data the order is:

$$O(n), \text{ where } n = \min(|s|, |t|)$$

d) Yes, the new metric should run slower because the addition of a domain check was redundant, it did not change the solution but added an additional n-dependent operation for every element comparison.

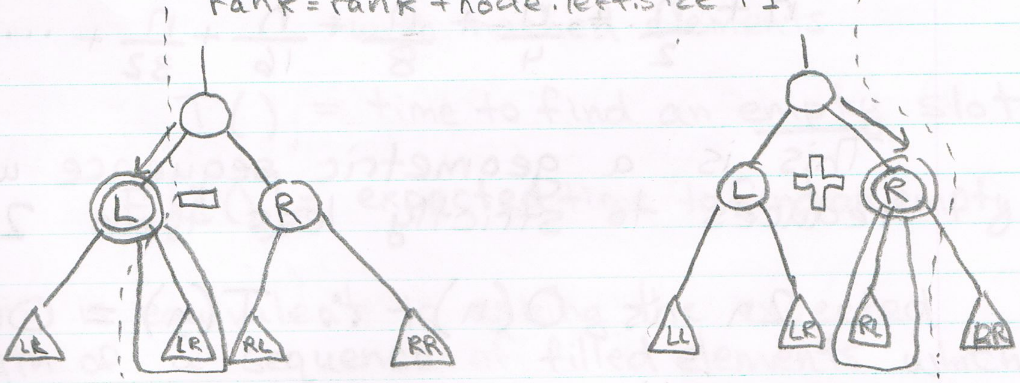
6.006
Scott Young

Problem Set 2 – Introduction to Algorithms

1) Implemented in bstselect.py

Algorithm description:

```
Set node to root
set rank = node.left's size
while node isn't None:
    if index == rank:
        return Node
    else if index < rank:
        node = node.left
        rank = rank - node.right.size - 1
    else
        node = node.right
        rank = rank + node.left.size + 1
```



LEFT TRAVERSAL

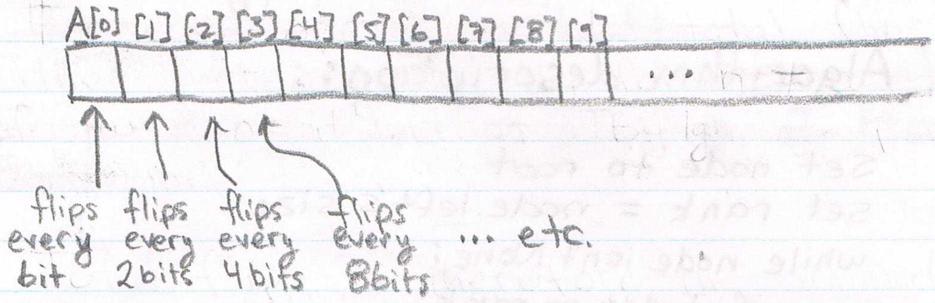
RIGHT TRAVERSAL

(dotted line is rank separator)

2) Amortized Analysis:

Prove $T(n) = O(n)$ for $T(n)$ being the number of operations to increment a binary counter n times.

Bits flip depending on the k-value for the k^{th} increment:



Therefore the total cost of k increments is a summation of the number of flips for each bit:

$$n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \frac{n}{16} + \frac{n}{32} + \cdots$$

This is a geometric sequence which reduces to strictly less than 2n.

$2n$ is $O(n)$ $\therefore$ $T(n) = O(n)$ **QED**

3) a) $P(\text{chain of 3 after 3 hashes}) = \left(\frac{1}{m}\right)^3$

Assuming the keys are independent, each has a probability of hashing to one locatic of $1/m$, the probability of 3 hashes occurin is $1/m \cdot 1/m \cdot 1/m$ or $(1/m)^3$

b) This probability is also $(1/m)^3$ because it amounts to the same thing, slot n is hashed 3 times in a row.

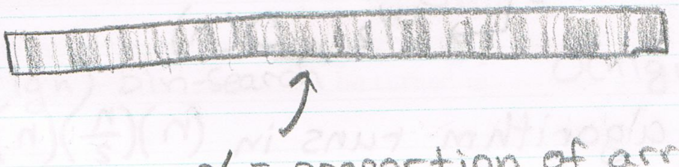
c) Load Factor $= \alpha = 1 - \frac{1}{\log n}$

$\alpha = n/m =$ ratio of elements to slots

$T(\text{unsuccessful search}) = 1 + \underset{\overset{\shortparallel}{\alpha}}{\text{average}}$ chain lengt

$$\boxed{T(n) = 2 - \frac{1}{\lg n}}$$

d)



$\alpha =$ proportion of array filled with hashed elements

$T() =$ time to find an <u>empty</u> slot

$E[T()] =$ expected time to find an empty sl

This is equivalent to asking the expected length of a sequence of filled elements which has a proportion α.

$$E[T(n)] = i, \quad \underbrace{\alpha^i = 0.5}_{\text{This is the probability where a longer and shorter subsequen are equal}}$$

$$\left(1 - \frac{1}{\lg n}\right)^i = 0.5 \qquad i = \frac{0.5}{\lg(1 - \frac{1}{\lg n})}$$

$$i \lg\left(1 - \frac{1}{\lg n}\right) = 0.5 \qquad E[T(n)] = \frac{1}{2\lg(1 - 1/\lg}$$

4)a) Ben's algorithm runs 4 nested loops:

1) for s_start in range(0, len(s))
   ↳ This is $O(n)$
2) for s_end in range(s-start, len(s))
   ↳ This has an amortized cost of $n/2$, also $O(n)$
3) for t_start in range(0, len(t))
   ↳ This is $O(n)$
4) " "
   ↳ $n/2$, $O(n)$

The algorithm runs in $(n)(\frac{n}{2})(n)(\frac{n}{2}) = \frac{n^4}{4}$ or O
time.

b) Alyssa's algorithm requires 2 nested for loops, plus a call to the Python <u>in</u> operator for lists:

1) for length in range(min(|s|, |t|))
   ↳ $O(n)$

2) for s_start ... $\longrightarrow O(n)$
   for t_start ... $\longrightarrow O(n)$ $+ 2*O(n) = O(n)$

3) if current in s_substrings
   ↳ I will assume this uses linear searching and therefore is $O(n)$

Therefore the total algorithm runs in $O(n^3)$ time.

c) Implemented in substring2.py

Initial (Unmodified)	$O(n^2 lg n)$ modification
T(100) = 0.0003719	0.0003859
T(1000) = 0.028105	0.00869703
T(10000) = 3.7917039	0.45809197
T(100000) = 204.160919	180.81293988

d) Implemented in substring4.py using Rabin-Karp:

	$O(n^2 lg n)$ bin-search	$O(n lg n)$ Rabin Karp
T(100)		
T(1000)		
T(10000)		
T(100000)		

6.006
Scott You

Problem Set 3 - Introduction to Algorithms

1) Implemented in heap_delete.py

Code uses first decrease_key which sets the index to $-\infty$ and pushes it to the top (maintaining minheap invariant). Then, the extract_min function is used to remove the element. Both operations run in $O(\lg n)$ so the combined function is also $O(\lg n)$

2) a) If the MPQ were implemented with a max heap, then m operations would have a worst-case time of $O(m \lg m)$ since extract and insert would each have $O(\lg m)$ time in the worst case.

b) Here, a length k array can be generated, every insert operation would check against a stored max value, updating only if the value $\leq$ max. Updates are done by setting MPQ[value] += 1. Extract max removes one from the MPQ[max], and if this is now zero walks back down the array until it can find a new max.

The process takes $O(m+k)$ for m operations because the total cost of all walk-backs to find a new max is at most k, since the max can only decrease.

3) a) Detect collisions is $O(n^2)$ since every ball is compared against every other. (Upper triangle in a square matrix)

b) Sorting cost $= O(n)$

Collision Check $= O(mr^2)$ where $r = E[\#\text{ of balls}/\text{b}$ and $m = \#$ of bins

In the worst case (all balls in the same bin) thi performs the same as the original algorith

On the average case, however, $r = \dfrac{n}{(400\sqrt{n})^2/256^2}$

$$r = \frac{n}{\frac{160000\,n}{65536}} \doteq \frac{1}{2.44}$$

Since $m \doteq 2.44n$, this means the collision check should take $O(n)$ time to process, albeit with higher constant coefficients.

c) Implemented in detection.py and teste

6.006
Scott You

PROBLEM SET 4: Introduction to Algorithms

1) Give an $O(V+E)$ time algorithm for separating a graph into connected components.

First, create a set of all vertices, ordered by some easy process (such as position in array or matrix). Next, go through every edge. Compare the two nodes incident to the edge, keep the smaller of the two and delete (or mark) the larger one. In the end, the graph will have only c unmarked nodes which are the smallest nodes in eac connected component.

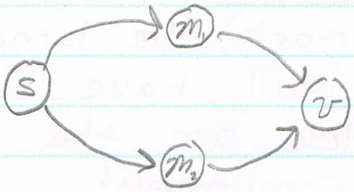
If the purpose was to partition the graph, then instead of marking, nodes could be adde to a list containing all connected nodes and stored by the "min" node.

Since only one pass through each node and each edge is required, this algorithm requires $O(V+E)$ time.

2) a) v is part of a cycle if a path connects it with itself. To implement this algorithm in $O(V+E)$ time, run a BFS starting at v never color the node v, so future edge checks can loop back. Then if v would be colored a second time, we can stop and return true. If we reach the end of all nodes, we can return false

b) Ben's algorithm works for undirected graphs because going backwards from an edge is always possible, therefore the only graphs without cycles are trees which never repeat nodes in a BFS.

In a digraph this process would produce false positives. Take the simple example belo



There is no cycle here because v has no outgoing edges, yet it <u>is</u> seen twice from m_1 and m_2.

c) First, we conduct a DFS, but also give a second mark to each node (*) when it is discovered and remove this mark whenever we backtrack from a black node. This (*) has the property of saying which nodes lie in a direct path from s, the starting node and v, the current vertex.

Cycles can then be detected as occurring whenever a node finds a (*) node during its search.

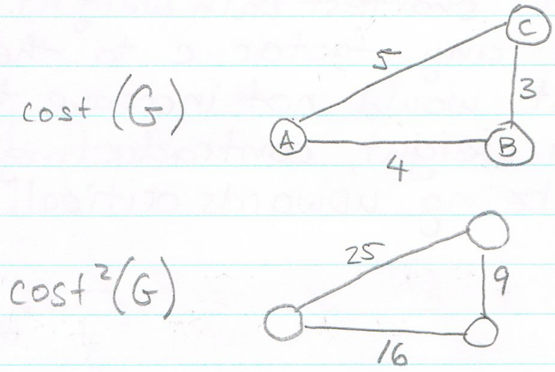
3) Implemented in level.py and solver.py

6.006
Scott Young

PROBLEM SET 5: INTRODUCTION TO ALGORITHMS

1) a) False. This is only true if the graph contains no negative edge weight cycles, as this will leave the shortest path undefined, but will return an answer nonetheless with this modification.

b) False. Consider the following graph:



In this example, the path AC is strictly short than ABC, yet they have the same length with all edge lengths squared.

c) True. This algorithm will return the largest simple path, in an acyclic graph. The relate problem of finding a longest simple path with cycles is not solvable with this algorithm and is in NP-complete.

2) a) Proof by Contradiction:

Suppose edge (u,v) were downward critica and it was not on a shortest path betwee s and v. This means the total path weight including (u,v) is strictly more than the shortest path weight by some factor c. We could then subtract some amount ac, $0<a<1$ from (u,v) and it would still be longer than the shortest path which contradicts the definition of being downward critical. QED

The reverse is also true as [illegible] in a shortest path weight must reduce the total weight, making it [illegible]

b) Claim: All upward critical edges (u, v) are on the shortest path between s and v.

Proof by Contradiction:

Again, suppose we had an upward critical path that weren't on the shortest path. This means the path including (u, v) is strictly larger than the shortest path weight. Here, we could add any factor c to the weight of (u, v) and it would not increase the shortest path weight, contradicting the principle of being upwards critical.

c)

3) Implemented in dijkstra.py

6.006
Scott Youn

PROBLEM SET 6: Introduction to Algorithms

1) Implemented in fib.py

2) a) The set of subproblems is the ways of making change with (1, 2, 3, ..., C-1) as the amounts of change.

b) Let SCL(C) represent the function for the list of the shortest list of change, given an amount of change, C.

$$SCL(c) = \min\begin{pmatrix} SCL(c-d_1), \\ SCL(c-d_2), \\ \cdots \\ SCL(c-d_m) \end{pmatrix} + 1$$

c) The running time of this algorithm is

$O(mC)$ as there are C subproblem each requiring m (# of denomina calculations.

d) Implemented in change.py

3) a) The set of subproblems are the longest increasing subsequences, ending at index $i \in 0,1,.$ The LIS of the first i elements is necessarily 1+the maximum of the LIS which end at j, which is such that $j<i$, $x[i]<x[j]$, given X as the sequen

b) Let q_i be the LIS which ends at element i, the recurrence relation is therefore:

$$q_i = \max \text{length}\left(q_j \mid j<i, x_j<x_i \right) + 1$$

From this the true LIS is simply:

$$LIS = \max \text{length}\left(q_i \mid i \in \{0,1,2,\dots,n\} \right)$$

c) This algorithm requires $O(n^2)$ time because there are n problems each having n subproble

d) Implemented in progress.py.

4) Implemented in ResizeableImage.py

↳ Unfortunately, I couldn't get PIL working so this code is untested and may contain bugs.