
Table of Contents

Introduction	1.1
--------------	-----

I Foundations

1 The Role of Algorithms in Computing	2.1
1.1 Algorithms	2.1.1
1.2 Algorithms as a technology	2.1.2
Problems	2.1.3
2 Getting Started	2.2
2.1 Insertion sort	2.2.1
2.2 Analyzing algorithms	2.2.2
2.3 Designing algorithms	2.2.3
Problems	2.2.4
3 Growth of Functions	2.3
3.1 Asymptotic notation	2.3.1
3.2 Standard notations and common functions	2.3.2
Problems	2.3.3
4 Divide-and-Conquer	2.4
4.1 The maximum-subarray problem	2.4.1
4.2 Strassen's algorithm for matrix multiplication	2.4.2
4.3 The substitution method for solving recurrences	2.4.3
4.4 The recursion-tree method for solving recurrences	2.4.4
4.5 The master method for solving recurrences	2.4.5
4.6 Proof of the master theorem	2.4.6
Problems	2.4.7
5 Probabilistic Analysis and Randomized Algorithms	2.5
5.1 The hiring problem	2.5.1
5.2 Indicator random variables	2.5.2
5.3 Randomized algorithms	2.5.3
5.4 Probabilistic analysis and further uses of indicator random variables	2.5.4

Problems	2.5.5
----------	-------

II Sorting and Order Statistics

6 Heapsort	3.1
6.1 Heaps	3.1.1
6.2 Maintaining the heap property	3.1.2
6.3 Building a heap	3.1.3
6.4 The heapsort algorithm	3.1.4
6.5 Priority queues	3.1.5
Problems	3.1.6
7 Quicksort	3.2
7.1 Description of quicksort	3.2.1
7.2 Performance of quicksort	3.2.2
7.3 A randomized version of quicksort	3.2.3
7.4 Analysis of quicksort	3.2.4
Problems	3.2.5
8 Sorting in Linear Time	3.3
8.1 Lower bounds for sorting	3.3.1
8.2 Counting sort	3.3.2
8.3 Radix sort	3.3.3
8.4 Bucket sort	3.3.4
Problems	3.3.5
9 Medians and Order Statistics	3.4
9.1 Minimum and maximum	3.4.1
9.2 Selection in expected linear time	3.4.2
9.3 Selection in worst-case linear time	3.4.3
Problems	3.4.4

III Data Structures

10 Elementary Data Structures	4.1
10.1 Stacks and queues	4.1.1
10.2 Linked lists	4.1.2

10.3 Implementing pointers and objects	4.1.3
10.4 Representing rooted trees	4.1.4
Problems	4.1.5
11 Hash Tables	4.2
11.1 Direct-address tables	4.2.1
11.2 Hash tables	4.2.2
11.3 Hash functions	4.2.3
11.4 Open addressing	4.2.4
11.5 Perfect hashing	4.2.5
Problems	4.2.6
12 Binary Search Trees	4.3
12.1 What is a binary search tree?	4.3.1
12.2 Querying a binary search tree	4.3.2
12.3 Insertion and deletion	4.3.3
12.4 Randomly built binary search trees	4.3.4
Problems	4.3.5
13 Red-Black Trees	4.4
13.1 Properties of red-black trees	4.4.1
13.2 Rotations	4.4.2
13.3 Insertion	4.4.3
13.4 Deletion	4.4.4
Problems	4.4.5
14 Augmenting Data Structures	4.5
14.1 Dynamic order statistics	4.5.1
14.2 How to augment a data structure	4.5.2
14.3 Interval trees	4.5.3
Problems	4.5.4

IV Advanced Design and Analysis Techniques

15 Dynamic Programming	5.1
15.1 Rod cutting	5.1.1

15.2 Matrix-chain multiplication	5.1.2
15.3 Elements of dynamic programming	5.1.3
15.4 Longest common subsequence	5.1.4
15.5 Optimal binary search trees	5.1.5
Problems	5.1.6
16 Greedy Algorithm	5.2
16.1 An activity-selection problem	5.2.1
16.2 Elements of the greedy strategy	5.2.2
16.3 Huffman codes	5.2.3
16.4 Matroids and greedy methods	5.2.4
16.5 A task-scheduling problem as a matroid	5.2.5
Problems	5.2.6
17 Amortized Analysis	5.3
17.1 Aggregate analysis	5.3.1
17.2 The accounting method	5.3.2
17.3 The potential method	5.3.3
17.4 Dynamic tables	5.3.4
Problems 1	5.3.5
Problems 2	5.3.6

V Advanced Data Structures

18 B-Trees	6.1
18.1 Definition of B-trees	6.1.1
18.2 Basic operations on B-trees	6.1.2
18.3 Deleting a key from a B-tree	6.1.3
Problems	6.1.4
19 Fibonacci Heaps	6.2
19.1 Structure of Fibonacci heaps	6.2.1
19.2 Mergeable-heap operations	6.2.2
19.3 Decreasing a key and deleting a node	6.2.3
19.4 Bounding the maximum degree	6.2.4
Problems	6.2.5
20 van Emde Boas Trees	6.3

20.1 Preliminary approaches	6.3.1
20.2 A recursive structure	6.3.2
20.3 The van Emde Boas tree	6.3.3
Problems	6.3.4
21 Data Structures for Disjoint Sets	6.4
21.1 Disjoint-set operations	6.4.1
21.2 Linked-list representation of disjoint sets	6.4.2
21.3 Disjoint-set forests	6.4.3
21.4 Analysis of union by rank with path compression	6.4.4
Problems	6.4.5

VI Graph Algorithms

22 Elementary Graph Algorithms	7.1
22.1 Representations of graphs	7.1.1
22.2 Breadth-first search	7.1.2
22.3 Depth-first search	7.1.3
22.4 Topological sort	7.1.4
22.5 Strongly connected components	7.1.5
Problems	7.1.6
23 Minimum Spanning Trees	7.2
23.1 Growing a minimum spanning tree	7.2.1
23.2 The algorithms of Kruskal and Prim	7.2.2
Problems	7.2.3
24 Single-Source Shortest Paths	7.3
24.1 The Bellman-Ford algorithm	7.3.1
24.2 Single-source shortest paths in directed acyclic graphs	7.3.2
24.3 Dijkstra's algorithm	7.3.3
24.4 Difference constraints and shortest paths	7.3.4
24.5 Proofs of shortest-paths properties	7.3.5
Problems	7.3.6
25 All-Pairs Shortest Paths	7.4
25.1 Shortest paths and matrix multiplication	7.4.1
25.2 The Floyd-Warshall algorithm	7.4.2

25.3 Johnson's algorithm for sparse graphs	7.4.3
Problems	7.4.4
26 Maximum Flow	7.5
26.1 Flow networks	7.5.1
26.2 The Ford-Fulkerson method	7.5.2
26.3 Maximum bipartite matching	7.5.3
26.4 Push-relabel algorithms	7.5.4
26.5 The relabel-to-front algorithm	7.5.5
Problems	7.5.6

VII Selected Topics

27 Multithreaded Algorithms	8.1
27.1 The basics of dynamic multithreading	8.1.1
27.2 Multithreaded matrix multiplication	8.1.2
27.3 Multithreaded merge sort	8.1.3
Problems	8.1.4
28 Matrix Operations	8.2
28.1 Solving systems of linear equations	8.2.1
28.2 Inverting matrices	8.2.2
28.3 Symmetric positive-definite matrices and least-squares approximation	8.2.3
Problems	8.2.4
29 Linear Programming	8.3
29.1 Standard and slack forms	8.3.1
29.2 Formulating problems as linear programs	8.3.2
29.3 The simplex algorithm	8.3.3
29.4 Duality	8.3.4
29.5 The initial basic feasible solution	8.3.5
Problems	8.3.6
30 Polynomials and the FFT	8.4
30.1 Representing polynomials	8.4.1
30.2 The DFT and FFT	8.4.2
30.3 Efficient FFT implementations	8.4.3
Problems	8.4.4

31 Number-Theoretic Algorithms	8.5
31.1 Elementary number-theoretic notions	8.5.1
31.2 Greatest common divisor	8.5.2
31.3 Modular arithmetic	8.5.3
31.4 Solving modular linear equations	8.5.4
31.5 The Chinese remainder theorem	8.5.5
31.6 Powers of an element	8.5.6
31.7 The RSA public-key cryptosystem	8.5.7
31.8 Primality testing	8.5.8
31.9 Integer factorization	8.5.9
Problems	8.5.10
32 String Matching	8.6
32.1 The naive string-matching algorithm	8.6.1
32.2 The Rabin-Karp algorithm	8.6.2
32.3 String matching with finite automata	8.6.3
32.4 The Knuth-Morris-Pratt algorithm	8.6.4
Problems	8.6.5
33 Computational Geometry	8.7
33.1 Line-segment properties	8.7.1
33.2 Determining whether any pair of segments intersects	8.7.2
33.3 Finding the convex hull	8.7.3
33.4 Finding the closest pair of points	8.7.4
Problems	8.7.5
34 NP-Completeness	8.8
34.1 Polynomial time	8.8.1
34.2 Polynomial-time verification	8.8.2
34.3 NP-completeness and reducibility	8.8.3
34.4 NP-completeness proofs	8.8.4
34.5 NP-complete problems	8.8.5
Problems	8.8.6
35 Approximation Algorithms	8.9
35.1 The vertex-cover problem	8.9.1
35.2 The traveling-salesman problems	8.9.2
35.3 The set-covering problem	8.9.3

35.4 Randomization and linear programming	8.9.4
35.5 The subset-sum problem	8.9.5
Problem	8.9.6

Introduction

 Fork 48  Star 85  Watch 6  Follow 134

build passing issues 0 open closed issues 9 pull requests 0 open closed pull requests 8 closed
contributors 7

Some exercises and problems in ***Introduction to Algorithms (CLRS)*** 3rd edition.

1 The Role of Algorithms in Computing

- 1.1 Algorithms
- 1.2 Algorithms as a technology
- Problems

1.1 Algorithms

1.1-1

Give a real-world example that requires sorting or a real-world example that requires computing convex hull.

- Sorting: browse the games with ascending prices on Steam.
- Convex hull: computing the diameter of set of points.

1.1-2

Other than speed, what other measures of efficiency might one use in a real-world setting?

Memory efficiency and coding efficiency.

1.1-3

Select a data structure that you have seen previously, and discuss its strengths and limitations.

Linked-List:

- Strengths: insertion and deletion.
- Limitations: random access.

1.1-4

How are the shortest-path and traveling-salesman problems given above similar? How are they different?

- Similar: finding path with shortest distance.
- Different: traveling-salesman has more constraints.

1.1-5

Come up with a real-word problem in which only the best solution will do. Then come up with one in which a solution that is "approximately" the best is good enough.

- Best: find the GCD of two positive integer numbers.
- Approximately: find the solution of differential equations.

1.2 Algorithm as a technology

1.2-1

Give an example of an application that requires algorithmic content at the application level, and discuss the function of the algorithms involved.

Drive navigation.

1.2-2

Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size n , insertion sort runs in $8n^2$ steps, while merge sort runs in $64nlgn$ steps. For which values of n does insertion sort beat merge sort?

$$8n^2 < 64nlgn$$

$$2^n < n^8$$

$$n \leq 43$$

1.2-3

What is the smallest value of n such that an algorithm whose running time is $100n^2$ runs faster than an algorithm whose running time is 2^n on the same machine?

$$100n^2 < 2^n$$

$$n \geq 15$$

Problems

1-1 Comparison of running times

For each function $f(n)$ and time t in the following table, determine the largest size n of a problem that can be solved in time t , assuming that the algorithm to solve the problem takes $f(n)$ microseconds.

	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
lgn	2^{10^6}	$2^{6 \times 10^6}$	$2^{3.6 \times 10^9}$	$2^{8.64 \times 10^{10}}$	$2^{2.59 \times 10^{12}}$	$2^{3.15 \times 10^{13}}$	$2^{3.15 \times 10^{15}}$
\sqrt{n}	10^{12}	3.6×10^{15}	1.3×10^{19}	7.46×10^{21}	6.72×10^{24}	9.95×10^{26}	9.95×10^{30}
n	10^6	6×10^7	3.6×10^9	8.64×10^{10}	2.59×10^{12}	3.15×10^{13}	3.15×10^{15}
$nlg n$	6.24×10^4	2.8×10^6	1.33×10^8	2.76×10^9	7.19×10^{10}	7.98×10^{11}	6.86×10^{13}
n^2	1000	7745	60000	293938	1609968	5615692	56156922
n^3	100	391	1532	4420	13736	31593	146645
2^n	19	25	31	36	41	44	51
$n!$	9	11	12	13	15	16	17

```

import math

def log2(n):
    return math.log(n) / math.log(2)

complexities = [lambda n: math.sqrt(n),
                lambda n: n,
                lambda n: n * log2(n),
                lambda n: n ** 2,
                lambda n: n ** 3,
                lambda n: 2 ** n,
                lambda n: math.factorial(n)]

max_bound = [1e40, 1e20, 1e20, 1e10, 1e10, 100, 100]

times = [1000 * 1000,
         1000 * 1000 * 60,
         1000 * 1000 * 60 * 60,
         1000 * 1000 * 60 * 60 * 24,
         1000 * 1000 * 60 * 60 * 24 * 30,
         1000 * 1000 * 60 * 60 * 24 * 365,
         1000 * 1000 * 60 * 60 * 24 * 365 * 100]

print(' '.join(map(lambda v: '2^(' + '{:.2e}'.format(v) + ')', times)))

for k in range(len(complexities)):
    c = complexities[k]
    vals = []
    for t in times:
        l, r = 0, int(max_bound[k])
        max_n = 0
        while l <= r:
            mid = (l + r) // 2
            val = c(mid)
            if val == float('inf') or val > t:
                r = mid - 1
            else:
                l = mid + 1
                max_n = max(max_n, mid)
        vals.append(max_n)
    if k < 3:
        print(' '.join(map(lambda v: '{:.2e}'.format(v), vals)))
    else:
        print(' '.join(map(lambda v: str(int(math.floor(v))), vals)))

```

2 Getting Started

- 2.1 Insertion sort
- 2.2 Analyzing algorithms
- 2.3 Designing algorithms
- Problems

2.1 Insertion sort

2.1-1

Using Figure 2.2 as a model, illustrate the operation of INSERTION-SORT on the array $A = \langle 31, 41, 59, 26, 41, 58 \rangle$.

- $A = \langle 31, 41, 59, 26, 41, 58 \rangle$
- $A = \langle 31, 41, 59, 26, 41, 58 \rangle$
- $A = \langle 26, 31, 41, 59, 41, 58 \rangle$
- $A = \langle 26, 31, 41, 41, 59, 58 \rangle$
- $A = \langle 26, 31, 41, 41, 58, 59 \rangle$

2.1-2

Rewrite the INSERTION-SORT procedure to sort into nonincreasing instead of nondecreasing order.

```
def insertion_sort(a):  
    for j in range(1, len(a)):  
        key = a[j]  
        i = j - 1  
        while i >= 0 and a[i] < key:  
            a[i + 1] = a[i]  
            i -= 1  
        a[i + 1] = key
```

2.1-3

Consider the **searching problem**:

Input: A sequence of n numbers $A = \langle a_1, a_2, \dots, a_n \rangle$ and a value v .

Output: An index i such that $v = A[i]$ or the special value NIL if v does not appear in A .

Write pseudocode for **linear search**, which scans through the sequence, looking for v . Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the three necessary properties.

```
def linear_search(a, v):
    for i in range(len(a)):
        if a[i] == v:
            return i
    return None
```

2.1-4

Consider the problem of adding two n -bit binary integers, stored in two n -element arrays A and B . The sum of the two integers should be stored in binary form in an $(n + 1)$ -element array C . State the problem formally and write pseudocode for adding the two integers.

```
def add_binary(a, b):
    n = len(a)
    c = [0 for _ in range(n + 1)]
    carry = 0
    for i in range(n):
        c[i] = a[i] + b[i] + carry
        if c[i] > 1:
            c[i] -= 2
            carry = 1
        else:
            carry = 0
    c[n] = carry
    return c
```

2.2 Analyzing algorithms

2.2-1

Express the function $n^3/1000 - 100n^2 - 100n + 3$ in terms of Θ -notation.

$$\Theta(n^3)$$

2.2-2

Consider sorting n numbers stored in array A by first finding the smallest element of A and exchanging it with the element in $A[1]$. Then find the second smallest element of A , and exchange it with $A[2]$. Continue in this manner for the first $n - 1$ elements of A . Write pseudocode for this algorithm, which is known as **selection sort**. What loop invariant does this algorithm maintain? Why does it need to run for only the first $n - 1$ elements, rather than for all n elements? Give the best-case and worst-case running times of selection sort in Θ -notation.

```
def selection_sort(a):
    for i in range(len(a)):
        k = i
        for j in range(i + 1, len(a)):
            if a[j] < a[k]:
                k = j
        a[i], a[k] = a[k], a[i]
```

- Best-case: $\Theta(n^2)$
- Worst-case: $\Theta(n^2)$

2.2-3

Consider linear search again (see Exercise 2.1 - 3). How many elements of the input sequence need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case? What are the average-case and worst-case running times of linear search in Θ -notation? Justify your answers.

- Average: $n/2$ elements. $\Theta(n)$

- Worst: n elements. $\Theta(n)$

2.2-4

| How can we modify almost any algorithm to have a good best-case running time?

Adding special case.

2.3 Designing algorithms

2.3-1

Using Figure 2.4 as a model, illustrate the operation of merge sort on the array

$$A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$$

- $A = \langle 3, 41, 26, 52, 38, 57, 9, 49 \rangle$
- $A = \langle 3, 26, 41, 52, 9, 49, 38, 57 \rangle$
- $A = \langle 3, 9, 26, 38, 41, 49, 52, 57 \rangle$

2.3-2

Rewrite the MERGE procedure so that it does not use sentinels, instead stopping once either array L or R has had all its elements copied back to A and then copying the remainder of the other array back into A .

```
def merge_sort_sub(arr, l, r):
    if l >= r:
        return
    mid = (l + r) // 2
    merge_sort_sub(arr, l, mid)
    merge_sort_sub(arr, mid+1, r)
    arr_l = [arr[i] for i in range(l, mid+1)]
    arr_r = [arr[j] for j in range(mid+1, r+1)]
    i, j = 0, 0
    for k in range(l, r+1):
        if j == len(arr_r) or (i != len(arr_l) and arr_l[i] <= arr_r[j]):
            arr[k] = arr_l[i]
            i += 1
        else:
            arr[k] = arr_r[j]
            j += 1

def merge_sort(arr):
    merge_sort_sub(arr, 0, len(arr) - 1)
```

2.3-3

Use mathematical induction to show that when n is an exact power of 2, the solution of the recurrence

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

is $T(n) = n \lg n$.

- $T(2) = 2 = 2 \lg 2$
- Assume that $T(2^k) = 2^k \lg 2^k$, $k > 1$, then $T(2^{k+1}) = 2T(2^k) + 2^{k+1}$
 $= 2^{k+1} \lg 2^k + 2^{k+1} = 2^{k+1}(1 + \lg 2^k) = 2^{k+1}(\lg 2 + \lg 2^k)$
 $= 2^{k+1} \lg 2^{k+1}$

2.3-4

We can express insertion sort as a recursive procedure as follows. In order to sort $A[1..n]$, we recursively sort $A[1..n - 1]$ and then insert $A[n]$ into the sorted array $A[1..n - 1]$. Write a recurrence for the running time of this recursive version of insertion sort.

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

2.3-5

Referring back to the searching problem (see Exercise 2.1-3), observe that if the sequence A is sorted, we can check the midpoint of the sequence against v and eliminate half of the sequence from further consideration. The binary search algorithm repeats this procedure, halving the size of the remaining portion of the sequence each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is $\Theta(\lg n)$.

```

def binary_search(a, v):
    l, r = 0, len(a)-1
    while l <= r:
        mid = (l + r) // 2
        if a[mid] == v:
            return mid
        elif a[mid] < v:
            l = mid + 1
        else:
            r = mid - 1
    return None

```

$$T(n) = T(n/2) + C$$

2.3-6

Observe that the while loop of lines 5-7 of the INSERTION-SORT procedure in Section 2.1 uses a linear search to scan (backward) through the sorted subarray $A[1..j-1]$. Can we use a binary search (see Exercise 2.3 - 5) instead to improve the overall worst-case running time of insertion sort to $\Theta(n \lg n)$?

No, still has to move the elements for $\Theta(n)$ in each iteration.

2.3-7 *

Describe a $\Theta(n \lg n)$ -time algorithm that, given a set S of n integers and another integer x , determines whether or not there exist two elements in S whose sum is exactly x .

Sort the elements then:

```

def two_sum(a, x):
    l, r = 0, len(a)-1
    while l < r:
        if a[l] + a[r] == x:
            return True
        elif a[l] + a[r] < x:
            l += 1
        else:
            r -= 1
    return False

```


Problems

2-1 Insertion sort on small arrays in merge sort

Although merge sort runs in $\Theta(n \lg n)$ worst-case time and insertion sort runs in $\Theta(n^2)$ worst-case time, the constant factors in insertion sort can make it faster in practice for small problem sizes on many machines. Thus, it makes sense to **coarsen** the leaves of the recursion by using insertion sort within merge sort when subproblems become sufficiently small. Consider a modification to merge sort in which $n = k$ sublists of length k are sorted using insertion sort and then merged using the standard merging mechanism, where k is a value to be determined.

- a. Show that insertion sort can sort the n/k sublists, each of length k , in $\Theta(nk)$ worst-case time.

$$\Theta(k^2) \cdot n/k = \Theta(nk)$$

- b. Show how to merge the sublists in $\Theta(n \lg(n/k))$ worst-case time.

- Layers: $\lg(n/k)$
- Each: n

- c. Given that the modified algorithm runs in $\Theta(nk + n \lg(n/k))$ worst-case time, what is the largest value of k as a function of n for which the modified algorithm has the same running time as standard merge sort, in terms of Θ -notation?

Since $n \lg(n/k) \leq n \lg n$, thus $nk = n \lg n$, $k = \lg n$.

- d. How should we choose k in practice?

Profiling with large data set.

2-2 Correctness of bubblesort

Bubblesort is a popular, but inefficient, sorting algorithm. It works by repeatedly swapping adjacent elements that are out of order.

```
BUBBLESORT(A)
1 for i= 1 to A.length - 1
2     for j = A.length downto i + 1
3         if A[j] < A[j - 1]
4             exchange A[j] with A[j - 1]
```

a. Let A' denote the output of BUBBLESORT(A). To prove that BUBBLESORT is correct, we need to prove that it terminates and that

$$A'[1] \leq A'[2] \leq \dots \leq A'[n] \quad (2.3)$$

where $n = A.length$. In order to show that BUBBLESORT actually sorts, what else do we need to prove?

A' is a permutation of A .

b. State precisely a loop invariant for the for loop in lines 2–4, and prove that this loop invariant holds. Your proof should use the structure of the loop invariant proof presented in this chapter.

- Initialization: $A[1]$ is sorted
- Maintenance: Move the smallest element to the left
- Termination: $A[1..i]$ is sorted with the next smallest element in $A[i]$

c. Using the termination condition of the loop invariant proved in part (b), state a loop invariant for the for loop in lines 1–4 that will allow you to prove inequality (2.3). Your proof should use the structure of the loop invariant proof presented in this chapter.

- Initialization: $A[1..i-1]$ is sorted with smallest elements
- Maintenance: Move the next smallest element to $A[i]$ and $A[i - 1] \leq A[i]$
- Termination: (2.3)

d. What is the worst-case running time of bubblesort? How does it compare to the running time of insertion sort?

$$\Theta(n^2)$$

For average case insertion sort is better.

2-3 Correctness of Horner's rule

The following code fragment implements Horner's rule for evaluating a polynomial

$$\begin{aligned} P(x) &= \sum_{k=0}^n a_k x^k \\ &= a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + x a_n) \cdots)) \end{aligned}$$

given the coefficients a_0, a_1, \dots, a_n and a value for x :

```

1 y = 0
2 for i = n downto 0
3     y = ai + x * y

```

a. In terms of Θ -notation, what is the running time of this code fragment for Horner's rule?

$\Theta(n)$

b. Write pseudocode to implement the naive polynomial-evaluation algorithm that computes each term of the polynomial from scratch. What is the running time of this algorithm? How does it compare to Horner's rule?

```

def polynomial_evaluation(a, x):
    sum = 0
    for i in range(len(a)):
        sum += a[i] * x ** i
    return sum

```

$\Theta(n^2)$

c. Consider the following loop invariant:

At the start of each iteration of the for loop of lines 2–3,

$$y = \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k$$

Interpret a summation with no terms as equaling 0. Following the structure of the loop invariant proof presented in this chapter, use this loop invariant to show that, at

$$\text{termination, } y = \sum_{k=0}^n a_k x^k$$

- **Initialization:** $y = 0$

- **Maintenance:**
$$\begin{aligned} y &= a_i + x \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^k \\ &= a_i x^0 + \sum_{k=0}^{n-(i+1)} a_{k+i+1} x^{k+1} = a_i x^0 + \sum_{k=1}^{n-i} a_{k+i} x^k = \sum_{k=0}^{n-i} a_{k+i} x^k \end{aligned}$$
- **Termination:** $y = \sum_{k=0}^n a_k x^k$

d. Conclude by arguing that the given code fragment correctly evaluates a polynomial characterized by the coefficients a_0, a_1, \dots, a_n .

$$\sum y_i = P(x)$$

2-4 Inversions

Let $A[1..n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an *inversion* of A .

a. List the five inversions of the array $\langle 2, 3, 8, 6, 1 \rangle$.

- (2, 1)
- (3, 1)
- (8, 6)
- (8, 1)
- (6, 1)

b. What array with elements from the set $\{1, 2, \dots, n\}$ has the most inversions? How many does it have?

- Most: $\{n, n - 1, \dots, 1\}$
- How many: $\frac{n(n-1)}{2}$

c. What is the relationship between the running time of insertion sort and the number of inversions in the input array? Justify your answer.

Equal

d. Give an algorithm that determines the number of inversions in any permutation on n elements in $\Theta(n \lg n)$ worst-case time. (Hint: Modify merge sort.)

```
def count_inversion_sub(arr, l, r):
    if l >= r:
        return 0
    mid = (l + r) // 2
    cnt = count_inversion_sub(arr, l, mid) + count_inversion_sub(arr, mid+1, r)
    arr_l = [arr[i] for i in range(l, mid+1)]
    arr_l.append(1e100)
    arr_r = [arr[j] for j in range(mid+1, r+1)]
    arr_r.append(1e100)
    i, j = 0, 0
    for k in range(l, r+1):
        if arr_l[i] <= arr_r[j]:
            arr[k] = arr_l[i]
            i += 1
        else:
            arr[k] = arr_r[j]
            j += 1
            cnt += len(arr_l) - i - 1
    return cnt

def count_inversion(arr):
    return count_inversion_sub(arr, 0, len(arr) - 1)
```

3 Growth of Functions

- 3.1 Asymptotic notation
- 3.2 Standard notations and common functions
- Problems

3.1 Asymptotic notation

3.1-1

Let $f(n)$ and $g(n)$ be asymptotically nonnegative functions. Using the basic definition of Θ -notation, prove that $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

$$0.5 \cdot (f(n) + g(n)) \leq \max(f(n), g(n)) \leq 1 \cdot (f(n) + g(n))$$

3.1-2

Show that for any real constants a and b , where $b > 0$, $(n + a)^b = \Theta(n^b)$.

If $a > 0$, then $(n + a)^b < (n + n)^b = (2n)^b = 2^b \cdot n^b$;

If $a < 0$, let $a := -a$, then $(n - a)^b > (n - 0.5n)^b = (0.5n)^b = 0.5^b \cdot n^b$,

Since $0.5^b \cdot n^b \leq (n + a)^b \leq 2^b \cdot n^b$, therefore $(n + a)^b = \Theta(n^b)$.

3.1-3

Explain why the statement, "The running time of algorithm A is at least $O(n^2)$," is meaningless.

O is an upper bound, which means A could be $\Omega(1)$.

3.1-4

Is $2^{n+1} = O(2^n)$? Is $2^{2n} = O(2^n)$?

- $2^{n+1} = O(2^n)$?

$1 \cdot 2^n \leq 2^{n+1} \leq 2 \cdot 2^n$, thus $2^{n+1} = O(2^n)$.

- $2^{2n} = O(2^n)$?

$2^{2n} \leq c \cdot 2^n$

$2^n \leq c$ which is impossible, thus $2^{2n} \neq O(2^n)$.

3.1-5

Prove Theorem 3.1.

Theorem 3.1

For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

$f(n) = O(g(n))$ implies $0 \leq f(n) \leq c_2 g(n)$

$f(n) = \Omega(g(n))$ implies $0 \leq c_1 g(n) \leq f(n)$

Thus $c_1 g(n) \leq f(n) \leq c_2 g(n)$, $f(n) = \Theta(g(n))$, and vice versa.

3.1-6

Prove that the running time of an algorithm is $\Theta(g(n))$ if and only if its worst-case running time is $O(g(n))$ and its best-case running time is $\Omega(g(n))$.

Theorem 3.1

3.1-7

Prove that $o(g(n)) \cap \omega(g(n))$ is the empty set.

There is no $f(n)$ that $f(n) < g(n)$ and $f(n) > g(n)$.

3.1-8

We can extend our notation to the case of two parameters n and m that can go to infinity independently at different rates. For a given function $g(n, m)$, we denote by $O(g(n, m))$ the set of functions

$O(g(n, m)) = \{f(n, m) : \text{there exist positive constants } c, n_0, \text{ and } m_0 \text{ such that } 0 \leq f(n, m) \leq cg(n, m) \text{ for all } n \geq n_0 \text{ or } m \geq m_0\}$.

Give corresponding definitions for $\Omega(g(n, m))$ and $\Theta(g(n, m))$.

$\Omega(g(n, m)) = \{f(n, m) : \text{there exist positive constants } c, n_0, \text{ and } m_0 \text{ such that } 0 \leq cg(n, m) \leq f(n, m) \text{ for all } n \geq n_0 \text{ or } m \geq m_0\}$.

$\Theta(g(n, m)) = \{f(n, m) : \text{there exist positive constants } c_1, c_2, n_0, \text{ and } m_0 \text{ such that } c_1g(n, m) \leq f(n, m) \leq c_2g(n, m) \text{ for all } n \geq n_0 \text{ or } m \geq m_0\}$.

3.2 Standard notations and common functions

3.2-1

Show that if $f(n)$ and $g(n)$ are monotonically increasing functions, then so are the functions $f(n) + g(n)$ and $f(g(n))$, and if $f(n)$ and $g(n)$ are in addition nonnegative, then $f(n) \cdot g(n)$ is monotonically increasing.

- $f(n) + g(n)$

$$n \leq m$$

$$f(n) \leq f(m) \text{ and } g(n) \leq g(m)$$

$$f(n) + g(n) \leq f(m) + g(m)$$

- $f(g(n))$

$$n \leq m$$

$$f(n) \leq f(m)$$

$$g(f(n)) \leq g(f(m))$$

- $f(n) \cdot g(n)$

$$n \leq m$$

$$f(n) \leq f(m) \text{ and } g(n) \leq g(m)$$

$$f(n) \cdot g(n) \leq f(m) \cdot g(m)$$

3.2-2

Prove equation (3.16).

$$a^{\log_b c} = c^{\log_b a} \quad (3.16)$$

$$a^{\log_b c} = a^{\frac{\log_a c}{\log_a b}} = c^{\frac{1}{\log_a b}} = c^{\log_b a}$$

3.2-3

Prove equation (3.19). Also prove that $n! = \omega(2^n)$ and $n! = o(n^n)$.

$$\lg(n!) = \Theta(n \lg n) \quad (3.19)$$

- $\lg(n!) = \Theta(n \lg n)$

Use Stirling's approximation:

$$\begin{aligned} \lg(n!) &= \lg(\sqrt{2\pi n} \left(\frac{n}{e}\right)^n e^{\alpha n}) = \lg(\sqrt{2\pi n}) + \lg(\left(\frac{n}{e}\right)^n) + \lg(e^{\alpha n}) \\ &= \Theta(\lg \sqrt{n}) + \Theta(n \lg n) + \Theta(n) = \Theta(n \lg n) \end{aligned}$$

- $n! = \omega(2^n)$

$$n! = n \cdot (n-1) \cdots \cdot 1 \geq 4 \cdot 2 \cdots \cdot 2 \cdot 1 = 2^n$$

- $n! = o(n^n)$

$$n! = n \cdot (n-1) \cdots \cdot 1 \leq n \cdot n \cdots \cdot n = n^n$$

3.2-4 *

Is the function $\lceil \lg n \rceil!$ polynomially bounded? Is the function $\lceil \lg \lg n \rceil!$ polynomially bounded?

- $\lceil \lg n \rceil!$

$$\lceil \lg n \rceil! = \sqrt{2\pi \lg n} \left(\frac{\lg n}{e}\right)^{\lg n} e^{\alpha \lg n} = \Theta((\lg n)^{\lg n})$$

$$\lg \lceil \lg n \rceil! = \Theta(\lg n \lg \lg n)$$

$$\lg n^p = \Theta(\lg n)$$

$$\Theta(\lg n \lg \lg n) > \Theta(\lg n)$$

\therefore not bounded.

- $\lceil \lg \lg n \rceil!$

$$\lceil \lg \lg n \rceil! = \Theta((\lg \lg n)^{\lg \lg n})$$

$$\lg \lceil \lg \lg n \rceil! = \Theta(\lg \lg n \lg \lg \lg n) = o(\lg^2 \lg n)$$

$$\therefore \lg^b n = o(n^a)$$

$\therefore o(\lg^2 \lg n) = o(\lg n)$, is polynomially bounded.

3.2-5 *

Which is asymptotically larger: $\lg(\lg^* n)$ or $\lg^*(\lg n)$?

$$\lg(\lg^*(2^m)) \text{ and } \lg^*(\lg(2^m))$$

$$\lg(1 + \lg^* m) \text{ and } \lg^* m$$

$$\therefore \lg(x) < x$$

\therefore The right hand side is larger.

3.2-6

Show that the golden ratio ϕ and its conjugate $\hat{\phi}$ both satisfy the equation $x^2 = x + 1$.

$$\phi = \frac{1+\sqrt{5}}{2}$$

$$\phi^2 = \frac{6+2\sqrt{5}}{4} = \frac{1+\sqrt{5}}{2} + 1 = \phi + 1$$

$$\hat{\phi} = \frac{1-\sqrt{5}}{2}$$

$$\hat{\phi}^2 = \frac{6-2\sqrt{5}}{4} = \frac{1-\sqrt{5}}{2} + 1 = \hat{\phi} + 1$$

3.2-7

Prove by induction that the i th Fibonacci number satisfies the equality

$$F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}}$$

where ϕ is the golden ratio and $\hat{\phi}$ is its conjugate.

$$F_0 = 0, \frac{\phi^0 - \hat{\phi}^0}{\sqrt{5}} = 0$$

$$F_1 = 1, \frac{\phi - \hat{\phi}}{\sqrt{5}} = 1$$

Suppose $F_{i-2} = \frac{\phi^{i-2} - \hat{\phi}^{i-2}}{\sqrt{5}}$ and $F_{i-1} = \frac{\phi^{i-1} - \hat{\phi}^{i-1}}{\sqrt{5}}$,

$$F_i = F_{i-2} + F_{i-1} = \frac{1}{\sqrt{5}}(\phi^{i-2} - \hat{\phi}^{i-2} + \phi^{i-1} - \hat{\phi}^{i-1})$$

Based on the previous exercise,

$$\phi^{i-2} + \phi^{i-1} = \phi^{i-2}(1 + \phi) = \phi^{i-2}\phi^2 = \phi^i$$

$$\therefore F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}}$$

3.2-8

Show that $k \ln k = \Theta(n)$ implies $k = \Theta(n/\ln n)$.

$$c_1 n \leq k \ln k \leq c_2 n$$

$$\ln(c_1 n) \leq \ln(k \ln k) \leq \ln(c_2 n)$$

$$\ln c_1 + \ln n \leq \ln k + \ln \ln k \leq \ln c_2 + \ln n$$

$$\therefore \ln k + \ln \ln k \leq 2 \ln k \geq \ln c_1 + \ln n$$

$$\therefore \frac{\ln k}{\ln n} \geq \frac{1}{2}$$

$$\therefore \ln k + \ln \ln k \geq \ln k \leq \ln c_2 + \ln n$$

$$\therefore \frac{\ln k}{\ln n} \leq 1$$

$$\therefore c_1 n \leq k \ln k \leq c_2 n$$

$$\therefore \frac{c_1 n}{\ln n} \leq \frac{k \ln k}{\ln n} \leq \frac{c_2 n}{\ln n}$$

$$\therefore \frac{c_1 n}{\ln n} \leq \frac{k \ln k}{\ln n} \leq k \quad \text{and} \quad \frac{c_2 n}{\ln n} \geq \frac{k \ln k}{\ln n} \geq \frac{1}{2}k$$

$$\therefore c_1 \frac{n}{\ln n} \leq k \leq (2c_2) \frac{n}{\ln n}$$

$$\therefore k = \Theta(n/\ln n)$$

Problems

3-1 Asymptotic behavior of polynomials

Let

$$p(n) = \sum_{i=0}^d a_i n^i,$$

where $a_d > 0$, be a degree- d polynomial in n , and let k be a constant. Use the definitions of the asymptotic notations to prove the following properties.

a. If $k \geq d$, then $p(n) = O(n^k)$

$$p(n) = \sum_{i=0}^d a_i n^i \leq \sum_{i=0}^d a_i n^k$$

b. If $k \leq d$, then $p(n) = \Omega(n^k)$

$$p(n) = \sum_{i=0}^d a_i n^i \geq \sum_{i=k}^d a_i n^i$$

c. If $k = d$, then $p(n) = \Theta(n^k)$

$$p(n) = O(n^k) \text{ and } p(n) = \Theta(n^k)$$

d. If $k > d$, then $p(n) = o(n^k)$

$$p(n) = \sum_{i=0}^d a_i n^i < \sum_{i=0}^d a_i n^k$$

e. If $k < d$, then $p(n) = \omega(n^k)$

$$p(n) = \sum_{i=0}^d a_i n^i > \sum_{i=k}^d a_i n^i$$

3-2 Relative asymptotic growths

Indicate, for each pair of expressions (A, B) in the table below, whether A is O , o , Ω , ω , or Θ of B . Assume that $k \leq 1$, $\epsilon > 0$, and $c > 1$ are constants. Your answer should be in the form of the table with "yes" or "no" written in each box.

A	B	O	o	Ω	ω	Θ
$\lg^k n$	n^ϵ	yes	yes	no	no	no
n^k	c^n	yes	yes	no	no	no
\sqrt{n}	$n^{\sin n}$	no	no	no	no	no
2^n	$2^{n/2}$	no	no	yes	yes	no
$n^{\lg c}$	$c^{\lg n}$	yes	no	yes	no	yes
$\lg(n!)$	$\lg(n^n)$	yes	no	yes	no	yes

3-3 Ordering by asymptotic growth rates

a. Rank the following functions by order of growth; that is, find an arrangement g_1, g_2, \dots, g_{30} of the functions satisfying $g_1 = \Omega(g_2), g_2 = \Omega(g_3), \dots, g_{29} = \Omega(g_{30})$. Partition your list into equivalence classes such that functions $f(n)$ and $g(n)$ are in the same class if and only if $f(n) = \Theta(g(n))$.

$$\begin{aligned} 2^{2^{n+1}} &> 2^{2^n} > (n+1)! > n! > e^n > n \cdot 2^n > 2^n > (\frac{3}{2})^n > n^{\lg \lg n} = (\lg n)^{\lg n} \\ (\lg n)! &> n^3 n^2 = 4^{\lg n} > n \ln n > \lg(n!) > n = 2^{\lg n} > (\sqrt{2})^{\lg n} > 2^{\sqrt{2 \lg n}} \\ > \lg^2 n &> \ln n > \sqrt{\lg n} > \ln \ln n > 2^{\lg^* n} > \lg^*(\lg n) = \lg^* n > \lg(\lg^* n) > 1 \\ &= n^{1/\lg n} \end{aligned}$$

b. Give an example of a single nonnegative function $f(n)$ such that for all functions $g_i(n)$ in part (a), $f(n)$ is neither $O(g_i(n))$ nor $\Omega(g_i(n))$.

$$n^{\sin n}$$

3-4 Asymptotic notation properties

Let $f(n)$ and $g(n)$ be asymptotically positive functions. Prove or disprove each of the following conjectures.

a. $f(n) = O(g(n))$ implies $g(n) = O(f(n))$.

False, $f(n) = 1, g(n) = n$

b. $f(n) + g(n) = \Theta(\min(f(n), g(n)))$

False, $f(n) = 1, g(n) = n$

c. $f(n) = O(g(n))$ implies $\lg(f(n)) = O(\lg(g(n)))$, where $\lg(g(n)) \geq 1$ and $f(n) \geq 1$ for all sufficiently large n .

True.

$$f(n) \leq cg(n)$$

$$\lg(f(n)) \leq \lg c + \lg g(n) = O(\lg g(n))$$

d. $f(n) = O(g(n))$ implies $2^{f(n)} = O(2^{g(n)})$

False, $f(n) = 2n, g(n) = n$

e. $f(n) = O((f(n))^2)$

False, $f(n) = 1/n$

f. $f(n) = O(g(n))$ implies $g(n) = \Omega(f(n))$

True.

$$f(n) \leq cg(n)$$

$$g(n) \geq \frac{1}{c}f(n)$$

g. $f(n) = \Theta(f(n/2))$

False, $f(n) = 4^n, f(n/2) = 2^n$

h. $f(n) + o(f(n)) = \Theta(f(n))$

False.

3-5 Variations on O and Ω

Some authors define Ω in a slightly different way than we do; let's use $\overset{\infty}{\Omega}$ (read "omega infinity") for this alternative definition. We say that $f(n) = \overset{\infty}{\Omega}(g(n))$ if there exists a positive constant c such that $f(n) \geq cg(n) \geq 0$ for infinitely many integers n .

3-6 Iterated functions

We can apply the iteration operator $*$ used in the \lg^* function to any monotonically increasing function $f(n)$ over the reals. For a given constant $c \in \mathbb{R}$, we define the iterated function f_c^* by

$$f_c^*(n) = \min\{i \geq 0 : f^{(i)}(n) \leq c\},$$

which need not be well defined in all cases. In other words, the quantity $f_c^*(n)$ is the number of iterated applications of the function f required to reduce its argument down to c or less.

For each of the following functions $f(n)$ and constants c , give as tight a bound as possible on $f_c^*(n)$.

$f(n)$	c	$f_c^*(n)$
$n - 1$	0	$\Theta(n)$
$\lg n$	1	$\Theta(\lg^* n)$
$n/2$	1	$\Theta(\lg n)$
$n/2$	2	$\Theta(\lg n)$
\sqrt{n}	2	$\Theta(\lg \lg n)$
\sqrt{n}	1	not converge
$n^{1/3}$	2	$\Theta(\log_3 \lg n)$
$n/\lg n$	2	$\omega(\lg \lg n)o(\lg n)$

4 Divide-and-Conquer

- 4.1 The maximum-subarray problem
- 4.2 Strassen's algorithm for matrix multiplication
- 4.3 The substitution method for solving recurrences
- 4.4 The recursion-tree method for solving recurrences
- 4.5 The master method for solving recurrences
- 4.6 Proof of the master theorem
- Problems

4.1 The maximum-subarray problem

4.1-1

What does FIND-MAXIMUM-SUBARRAY return when all elements of A are negative?
 $(\text{smallest_index}, \text{smallest_index}, \text{smallest_value})$

4.1-2

Write pseudocode for the brute-force method of solving the maximum-subarray problem. Your procedure should run in $\Theta(n^2)$ time.

```
def find_maximum_subarray(arr):
    sums = [0]
    for a in arr:
        sums.append(sums[-1] + a)
    max_sum = -1e100
    left_index = -1
    right_index = -1
    for i in range(len(arr)):
        for j in range(i, len(arr)):
            if sums[j + 1] - sums[i] > max_sum:
                max_sum = sums[j + 1] - sums[i]
                left_index = i
                right_index = j
    return left_index, right_index, max_sum
```

4.1-3

Implement both the brute-force and recursive algorithms for the maximumsubarray problem on your own computer. What problem size n_0 gives the crossover point at which the recursive algorithm beats the brute-force algorithm? Then, change the base case of the recursive algorithm to use the brute-force algorithm whenever the problem size is less than n_0 . Does that change the crossover point?

```

def find_max_crossing_subarray(arr, low, mid, high):
    left_sum = -1e100
    sum = 0
    for i in range(mid - 1, low - 1, -1):
        sum = sum + arr[i]
        if sum > left_sum:
            left_sum = sum
            max_left = i
    right_sum = -1e100
    sum = 0
    for j in range(mid, high):
        sum = sum + arr[j]
        if sum > right_sum:
            right_sum = sum
            max_right = j
    return max_left, max_right, left_sum + right_sum

def find_maximum_subarray(arr, low, high):
    if low >= high:
        return -1, -1, -1e100
    if low + 1 == high:
        return low, low, arr[low]
    mid = (low + high) // 2
    left_low, left_high, left_sum = find_maximum_subarray(arr, low, mid)
    right_low, right_high, right_sum = find_maximum_subarray(arr, mid, high)
    cross_low, cross_high, cross_sum = find_max_crossing_subarray(arr, low, mid, high)
    if left_sum >= right_sum and left_sum >= cross_sum:
        return left_low, left_high, left_sum
    if right_sum >= left_sum and right_sum >= cross_sum:
        return right_low, right_high, right_sum
    return cross_low, cross_high, cross_sum

```

4.1-4

Suppose we change the definition of the maximum-subarray problem to allow the result to be an empty subarray, where the sum of the values of an empty subarray is 0. How would you change any of the algorithms that do not allow empty subarrays to permit an empty subarray to be the result?

Return empty if the result is negative.

4.1-5

Use the following ideas to develop a nonrecursive, linear-time algorithm for the maximum-subarray problem. Start at the left end of the array, and progress toward subarray seen so far. Knowing a maximum subarray of $A[1 \dots j]$, extend the answer to find a maximum subarray ending at index $j + 1$ by using the following observation: a maximum subarray of $A[1 \dots j + 1]$ is either a maximum subarray of $A[1 \dots j]$ or a subarray $A[i \dots j + 1]$, for some $1 \leq i \leq j + 1$. Determine a maximum subarray of the form $A[i \dots j + 1]$ in constant time based on knowing a maximum subarray ending at index j .

```
def find_maximum_subarray(arr):
    max_sum = -1e100
    max_left, max_right = -1, -1
    sum = 0
    last_left = 0
    for i in range(len(arr)):
        sum += arr[i]
        if sum > max_sum:
            max_sum = sum
            max_left = last_left
            max_right = i
        if sum < 0:
            sum = 0
            last_left = i + 1
    return max_left, max_right, max_sum
```

4.2 Strassen's algorithm for matrix multiplication

4.2-1

Use Strassen's algorithm to compute the matrix product

$$\begin{pmatrix} 1 & 3 \\ 7 & 5 \end{pmatrix} \begin{pmatrix} 6 & 8 \\ 4 & 2 \end{pmatrix}.$$

Show your work.

$$S_1 = B_{12} - B_{22} = 8 - 2 = 6$$

$$S_2 = A_{11} + A_{12} = 1 + 3 = 4$$

$$S_3 = A_{21} + A_{22} = 7 + 5 = 12$$

$$S_4 = B_{21} - B_{11} = 4 - 6 = -2$$

$$S_5 = A_{11} + A_{22} = 1 + 5 = 6$$

$$S_6 = B_{11} + B_{22} = 6 + 2 = 8$$

$$S_7 = A_{12} - A_{22} = 3 - 5 = -2$$

$$S_8 = B_{21} + B_{22} = 4 + 2 = 6$$

$$S_9 = A_{11} - A_{21} = 1 - 7 = -6$$

$$S_{10} = B_{11} + B_{12} = 6 + 8 = 14$$

$$P_1 = A_{11} \cdot S_1 = 1 \times 6 = 6$$

$$P_2 = S_2 \cdot B_{22} = 4 \times 2 = 8$$

$$P_3 = S_3 \cdot B_{11} = 12 \times 6 = 72$$

$$P_4 = A_{22} \cdot S_4 = 5 \times -2 = -10$$

$$P_5 = S_5 \cdot S_6 = 6 \times 8 = 48$$

$$P_6 = S_7 \cdot S_8 = -2 \times 6 = -12$$

$$P_7 = S_9 \cdot S_{10} = -6 \times 14 = -84$$

$$C_{11} = P_5 + P_4 - P_2 + P_6 = 48 - 10 - 8 - 12 = 18$$

$$C_{12} = P_1 + P_2 = 8 + 6 = 14$$

$$C_{21} = P_3 + P_4 = 72 - 10 = 62$$

$$C_{22} = P_5 + P_1 - P_3 - P_7 = 48 + 6 - 72 + 84 = 66$$

$$\begin{pmatrix} 18 & 14 \\ 62 & 66 \end{pmatrix}$$

4.2-2

Write pseudocode for Strassen's algorithm.

```

def matrix_product_strassen_sub(a, b, r_low, r_high, c_low, c_high):
    n = r_high - r_low
    if n == 1:
        return [[a[r_low][c_low] * b[r_low][c_low]]]
    mid = n // 2
    r_mid = (r_low + r_high) // 2
    c_mid = (c_low + c_high) // 2
    s = [[[0 for _ in range(mid)] for _ in range(mid)] for _ in range(10)]
    for i in range(mid):
        for j in range(mid):
            s[0][i][j] = b[r_low + i][c_mid + j] - b[r_mid + i][c_mid + j]
            s[1][i][j] = a[r_low + i][c_low + j] + a[r_low + i][c_mid + j]
            s[2][i][j] = a[r_mid + i][c_low + j] + a[r_mid + i][c_mid + j]
            s[3][i][j] = b[r_mid + i][c_low + j] - b[r_low + i][c_low + j]
            s[4][i][j] = a[r_low + i][c_low + j] + a[r_mid + i][c_mid + j]
            s[5][i][j] = b[r_low + i][c_low + j] + b[r_mid + i][c_mid + j]
            s[6][i][j] = a[r_low + i][c_mid + j] - a[r_mid + i][c_mid + j]
            s[7][i][j] = b[r_mid + i][c_low + j] + b[r_mid + i][c_mid + j]
            s[8][i][j] = a[r_low + i][c_low + j] - a[r_mid + i][c_low + j]
            s[9][i][j] = b[r_low + i][c_low + j] + b[r_low + i][c_mid + j]
    p = [[[0 for _ in range(mid)] for _ in range(mid)] for _ in range(7)]
    for i in range(mid):
        for j in range(mid):
            for k in range(mid):
                p[0][i][j] += a[r_low + i][c_low + k] * s[0][k][j]
                p[1][i][j] += s[1][i][k] * b[r_mid + k][c_mid + j]
                p[2][i][j] += s[2][i][k] * b[r_low + k][c_low + j]
                p[3][i][j] += a[r_mid + i][c_mid + k] * s[3][k][j]
                p[4][i][j] += s[4][i][k] * s[5][k][j]
                p[5][i][j] += s[6][i][k] * s[7][k][j]
                p[6][i][j] += s[8][i][k] * s[9][k][j]
    c = [[0 for _ in range(n)] for _ in range(n)]
    for i in range(mid):
        for j in range(mid):
            c[r_low + i][c_low + j] = p[4][i][j] + p[3][i][j] - p[1][i][j] + p[5][i][j]
    ]
        c[r_low + i][c_mid + j] = p[0][i][j] + p[1][i][j]
        c[r_mid + i][c_low + j] = p[2][i][j] + p[3][i][j]
        c[r_mid + i][c_mid + j] = p[4][i][j] + p[0][i][j] - p[2][i][j] - p[6][i][j]
    ]
    return c

def matrix_product_strassen(a, b):
    n = len(a)
    return matrix_product_strassen_sub(a, b, 0, n, 0, n)

```

4.2-3

How would you modify Strassen's algorithm to multiply $n \times n$ matrices in which n is not an exact power of 2? Show that the resulting algorithm runs in time $\Theta(n^{\lg 7})$.

Extend the matrix with zeros.

4.2-4

What is the largest k such that if you can multiply 3×3 matrices using k multiplications (not assuming commutativity of multiplication), then you can multiply $n \times n$ matrices in time $\Theta(n^{\lg 7})$? What would the running time of this algorithm be?

$$T(n) = kT(n/3) + O(n^2)$$

Running time: $\Theta(n^{\log_3 7})$

$$k \leq 3^{\lg 7} \approx 21.84986222490514$$

4.2-5

V. Pan has discovered a way of multiplying 68×68 matrices using 132,464 multiplications, a way of multiplying 70×70 matrices using 143,640 multiplications, and a way of multiplying 72×72 matrices using 155,424 multiplications. Which method yields the best asymptotic matrix-multiplication algorithm? How does it compare to Strassen's algorithm?

$$T_1 = \Theta(n^{\log_{68} 132464}) = \Theta(n^{2.7951284873613815})$$

$$T_2 = \Theta(n^{\log_{70} 143640}) = \Theta(n^{2.795122689748337})$$

$$T_3 = \Theta(n^{\log_{72} 155424}) = \Theta(n^{2.795147391093449})$$

$$T_2 < T_1 < T_3 < \Theta(n^{\lg 7})$$

4.2-6

How quickly can you multiply a $kn \times n$ matrix by an $n \times kn$ matrix, using Strassen's algorithm as a subroutine? Answer the same question with the order of the input matrices reversed.

$$\Theta(k^2 n^{\lg 7})$$

Reversed: $\Theta(kn^{\lg 7})$

4.2-7

Show how to multiply the complex numbers $a + bi$ and $c + di$ using only three multiplications of real numbers. The algorithm should take a, b, c , and d as input and produce the real component $ac - bd$ and the imaginary component $ad + bc$ separately.

$$P_1 = a \cdot (c - d)$$

$$P_2 = b \cdot (c + d)$$

$$P_3 = d \cdot (a - b)$$

Real component: $P_1 + P_3$

Image component: $P_2 + P_3$

4.3 The substitution method for solving recurrences

4.3-1

Show that the solution of $T(n) = T(n - 1) + n$ is $O(n^2)$.

Suppose $T(n) \leq n^2$

$$\begin{aligned} T(n) &\leq (n - 1)^2 + n \\ &= n^2 - 2n + 1 + n \\ &= n^2 - n + 1 \\ &\leq n^2 \end{aligned}$$

4.3-2

Show that the solution of $T(n) = T(\lceil n/2 \rceil) + 1$ is $O(\lg n)$.

Suppose $T(n) \leq c \lg(n - 1)$

$$\begin{aligned} T(n) &\leq c \lg(\lceil n/2 \rceil - a) + 1 \\ &\leq c \lg\left(\frac{n+1}{2} - a\right) + 1 \\ &= c \lg(n+1-2a) - c + 1 \\ &\leq c \lg(n-a) - c + 1 \quad (a \geq \frac{1}{3}) \\ &\leq c \lg(n-a) \quad (c \geq 1) \end{aligned}$$

4.3-3

We saw that the solution of $T(n) = 2T(\lfloor n/2 \rfloor) + n$ is $O(n \lg n)$. Show that the solution of this recurrence is also $\Omega(n \lg n)$. Conclude that the solution is $\Theta(n \lg n)$.

Suppose $T(n) \geq cn \lg n$

$$\begin{aligned}
 T(n) &\geq 2c(n/2) \lg(n/2) + n \\
 &= cn \lg n - cn + n \\
 &\geq cn \lg n \quad (c \leq 1)
 \end{aligned}$$

4.3-4

Show that by making a different inductive hypothesis, we can overcome the difficulty with the boundary condition $T(1) = 1$ for recurrence (4.19) without adjusting the boundary conditions for the inductive proof.

Suppose $T(n) \leq cn \lg n + n$

$$\begin{aligned}
 T(n) &\leq cn \lg n - 2n + n \\
 &\leq cn \lg n + n
 \end{aligned}$$

When $n = 1$, $T(n) = 0 + 1 = 1$.

4.3-5

Show that $\Theta(n \lg n)$ is the solution to the "exact" recurrence (4.3) for merge sort.

We know $T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n$ is $O(n \lg n)$.

Based on 4.3-3, $T(n)$ is $\Omega(n \lg n)$.

Therefore, $T(n)$ is $\Theta(n \lg n)$.

4.3-6

Show that the solution to $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$ is $O(n \lg n)$.

Suppose $T(n) \leq c(n - a) \lg(n - a)$

$$\begin{aligned}
 T(n) &\leq c(n + 34 - 2a) \lg(n + 34 - 2a) \\
 &\leq c(n - a) \lg(n - a) \quad (a \geq 34)
 \end{aligned}$$

4.3-7

Using the master method in Section 4.5, you can show that the solution to the recurrence $T(n) = 4T(n/3) + N$ is $T(n) = \Theta(n^{\log_3 4})$. Show that a substitution proof with the assumption $T(n) \leq n^{\log_3 4}$ fails. Then show how to subtract off a lower-order term to make a substitution proof work.

Suppose $T(n) \leq cn^{\log_3 4}$

$$T(n) \leq cn^{\log_3 4} + n$$

Suppose $T(n) \leq cn^{\log_3 4} - an$

$$\begin{aligned} T(n) &\leq cn^{\log_3 4} + (1 - \frac{4}{3})n \\ &\leq cn^{\log_3 4} - an \quad (a \geq 3) \end{aligned}$$

4.3-8

Using the master method in Section 4.5, you can show that the solution to the recurrence $T(n) = 4T(n/2) + n$ is $T(n) = \Theta(n^2)$. Show that a substitution proof with the assumption $T(n) = cn^2$ fails. Then show how to subtract off a lower-order term to make a substitution proof work.

Suppose $T(n) \leq cn^2$

$$T(n) \leq cn^2 + n$$

Suppose $T(n) \leq cn^2 - an$

$$\begin{aligned} T(n) &\leq cn^2 + (1 - 2a)n \\ &\leq cn^2 - an \quad (a \geq \frac{1}{3}) \end{aligned}$$

4.3-9

Solve the recurrence $T(n) = 3T(\sqrt{n}) + \log n$ by making a change of variables.

Your solution should be an integral.

Let $n = 2^m$

$$T(2^m) = 3T(2^{m/2}) + m$$

$$S(m) = 3S(m/2) + m$$

$$S(m) = \Theta(m^{\lg 3})$$

$$T(n) = \Theta(\lg^{\lg 3} n)$$

4.4 The recursion-tree method for solving recurrences

4.4-1

Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = 3T(\lfloor n/2 \rfloor) + n$. Use the substitution method to verify your answer.

$$\begin{aligned} T(n) &= \sum_{i=0}^{\lg n - 1} \left(\frac{3}{2}\right)^i n + \Theta(n^{\lg 3}) \\ &= \frac{(3/2)^{\lg n} - 1}{(3/2) - 1} n + \Theta(n^{\lg 3}) \\ &= 2n^{\lg 3} - 2n + \Theta(n^{\lg 3}) \\ &= \Theta(n^{\lg 3}) \end{aligned}$$

4.4-2

Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = T(n/2) + n^2$. Use the substitution method to verify your answer.

$$\begin{aligned} T(n) &= \sum_{i=0}^{\lg n - 1} \left(\frac{1}{2}\right)^i n^2 + \Theta(1) \\ &= \frac{(1/2)^{\lg n} - 1}{(1/2) - 1} n^2 + \Theta(1) \\ &= 2n^2 - \frac{2}{n} + \Theta(1) \\ &= \Theta(n^2) \end{aligned}$$

4.4-3

Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = 4T(n/2 + 2) + n$. Use the substitution method to verify your answer.

$$\begin{aligned}
 T(n) &= \sum_{i=0}^{\lg n - 1} (2^i n) + \sum_{i=0}^{\lg n - 1} (2^{2i+1}) + \Theta(n^2) \\
 &= \frac{2^{\lg n} - 1}{2 - 1} n + \Theta(n^2) \\
 &= n^2 - n + \Theta(n^2) \\
 &= \Theta(n^2)
 \end{aligned}$$

4.4-4

Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = 2T(n - 1) + 1$. Use the substitution method to verify your answer.

$$\begin{aligned}
 T(n) &= \sum_{i=0}^{n-1} (2^i) + \Theta(2^n) \\
 &= \frac{2^n - 1}{2 - 1} + \Theta(2^n) \\
 &= 2^n - 1 + \Theta(2^n) \\
 &= \Theta(2^n)
 \end{aligned}$$

4.4-5

Use a recursion tree to determine a good asymptotic upper bound on the recurrence $T(n) = T(n - 1) + T(n/2) + n$. Use the substitution method to verify your answer.

$$O(n^{\lg n})$$

4.4-6

Argue that the solution to the recurrence $T(n) = T(n/3) + T(2n/3) + cn$, where c is a constant, is $\Omega(n \lg n)$ by appealing to a recursion tree.

Shortest path is $\log_3 n$.

4.4-7

Draw the recursion tree for $T(n) = 4T(\lfloor n/2 \rfloor) + cn$, where c is a constant, and provide a tight asymptotic bound on its solution. Verify your bound by the substitution method.

$$\Theta(n^2)$$

4.4-8

Use a recursion tree to give an asymptotically tight solution to the recurrence $T(n) = T(n - a) + T(a) + cn$, where $a \geq 1$ and $c > 0$ are constants.

$$\begin{aligned} T(n) &= \sum_{i=0}^{n/a} (c(n - ai)) + cn \\ &= \Theta(n^2) \end{aligned}$$

4.4-9

Use a recursion tree to give an asymptotically tight solution to the recurrence $T(n) = T(\alpha n) + T((1 - \alpha)n) + cn$, where α is a constant in the range $0 < \alpha < 1$ and $c > 0$ is also a constant.

$$\begin{aligned} T(n) &= \sum_{i=0}^{\log_{1/\alpha} n} cn + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned}$$

4.5 The master method for solving recurrences

4.5-1

Use the master method to give tight asymptotic bounds for the following recurrences.

a. $T(n) = 2T(n/4) + 1$

$$\Theta(n^{\log_4 2}) = \Theta(\sqrt{n})$$

b. $T(n) = 2T(n/4) + \sqrt{n}$

$$\Theta(\sqrt{n} \lg n)$$

c. $T(n) = 2T(n/4) + n$

$$\Theta(n)$$

d. $T(n) = 2T(n/4) + n^2$

$$\Theta(n^2)$$

4.5-2

Professor Caesar wishes to develop a matrix-multiplication algorithm that is asymptotically faster than Strassen's algorithm. His algorithm will use the divide and conquer method, dividing each matrix into pieces of size $n/4 \times n/4$, and the divide and combine steps together will take $\Theta(n^2)$ time. He needs to determine how many subproblems his algorithm has to create in order to beat Strassen's algorithm. If his algorithm creates a subproblems, then the recurrence for the running time $T(n)$ becomes $T(n) = aT(n/4) + \Theta(n^2)$. What is the largest integer value of a for which Professor Caesar's algorithm would be asymptotically faster than Strassen's algorithm?

$$\log_4 a < \log_2 7$$

The largest a is 48.

4.5-3

Use the master method to show that the solution to the binary-search recurrence $T(n) = T(n/2) + \Theta(1)$ is $T(n) = \Theta(\lg n)$. (See Exercise 2.3-5 for a description of binary search.)

$$n^{\log_2 1} = 1$$

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

4.5-4

Can the master method be applied to the recurrence $T(n) = 4T(n/2) + n^2 \lg n$? Why or why not? Give an asymptotic upper bound for this recurrence.

No. $O(n^2 \lg n)$.

4.5-5 *

Consider the regularity condition $af(n/b) \leq cf(n)$ for some constant $c < 1$, which is part of case 3 of the master theorem. Give an example of constants $a \geq 1$ and $b > 1$ and a function $f(n)$ that satisfies all the conditions in case 3 of the master theorem except the regularity condition.

$$T(n) = T(n/2) + n(\sin(n - \pi/2) + 2)$$

4.6 Proof of the master theorem

4.6-1 *

Give a simple and exact expression for n_j in equation (4.27) for the case in which b is a positive integer instead of an arbitrary real number.

$$n_j = \left\lceil \frac{n}{b^j} \right\rceil$$

4.6-2 *

Show that if $f(n) = \Theta(n^{\log_b a} \lg^k n)$, where $k \geq 0$, then the master recurrence has solution $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$. For simplicity, confine your analysis to extract powers of b .

...

4.6-3 *

Show that case 3 of the master theorem is overstated, in the sense that the regularity condition $af(n/b) \leq cf(n)$ for some constant $c < 1$ implies that there exists a constant $\epsilon > 0$ such that $f(n) = \Omega(n^{\log_b a + \epsilon})$.

...

Problems

4-1 Recurrence examples

Give asymptotic upper and lower bounds for $T(n)$ in each of the following recurrences. Assume that $T(n)$ is constant for $n \leq 2$. Make your bounds as tight as possible, and justify your answers.

a. $T(n) = 2T(n/2) + n^4$

$$\Theta(n^4)$$

b. $T(n) = T(7n/10) + n$

$$\Theta(n)$$

c. $T(n) = 16T(n/4) + n^2$

$$\Theta(n^2 \lg n)$$

d. $T(n) = 7T(n/3) + n^2$

$$\Theta(n^2)$$

e. $T(n) = 7T(n/2) + n^2$

$$\Theta(n^{\lg 7})$$

f. $T(n) = 2T(n/4) + \sqrt{n}$

$$\Theta(\sqrt{n} \lg n)$$

g. $T(n) = T(n - 2) + n^2$

$$\Theta(n^3)$$

4-2 Parameter-passing costs

Throughout this book, we assume that parameter passing during procedure calls takes constant time, even if an N -element array is being passed. This assumption is valid in most systems because a pointer to the array is passed, not the array itself.

This problem examines the implications of three parameter-passing strategies:

1. An array is passed by pointer. Time = $\Theta(1)$.
2. An array is passed by copying. Time = $\Theta(N)$ where N is the size of the array.
3. An array is passed by copying only the subrange that might be accessed by the called procedure. Time D = $\Theta(q - p + 1)$ if the subarray $A[p \dots q]$ is passed.

a. Consider the recursive binary search algorithm for finding a number in a sorted array (see Exercise 2.3-5). Give recurrences for the worst-case running times of binary search when arrays are passed using each of the three methods above, and give good upper bounds on the solutions of the recurrences. Let N be the size of the original problem and n be the size of a subproblem.

1. $T(n) = T(n/2) + 1 = \Theta(\lg n)$
2. $T(n) = T(n/2) + N = \Theta(n \lg n)$
3. $T(n) = T(n/2) + n = \Theta(n)$

b. Redo part (a) for the MERGE-SORT algorithm from Section 2.3.1.

1. $T(n) = 2T(n/2) + n + c = 2T(n/2) + n = \Theta(n \lg n)$
2. $T(n) = 2T(n/2) + n + N = 2T(n/2) + N = \Theta(n^2)$
3. $T(n) = 2T(n/2) + n + n = 2T(n/2) + n = \Theta(n \lg n)$

4-3 More recurrence examples

Give asymptotic upper and lower bounds for $T(n)$ in each of the following recurrences. Assume that $T(n)$ is constant for sufficiently small n . Make your bounds as tight as possible, and justify your answers.

a. $T(n) = 4T(n/3) + n \lg n$

$$\Theta(n^{\log_3^4})$$

b. $T(n) = 3T(n/3) + n / \lg n$

For harmonic series:

$$\ln(n+1) \leq \int_1^{n+1} \frac{1}{t} dt \leq \sum_{i=1}^n \frac{1}{i} \leq 1 + \int_1^n \frac{1}{t} dt = 1 + \ln n$$

Therefore, harmonic series are $\Theta(\lg n)$

$$\begin{aligned} T(n) &= n \sum_{i=0}^{\log_3 n - 1} \frac{1}{\lg \frac{n}{3^i}} \\ &= \Theta(n \sum_{i=0}^{\log_3 n - 1} \frac{1}{\log_3 \frac{n}{3^i}}) \\ &= \Theta(n \sum_{i=1}^{\log_3 n} \frac{1}{i}) \\ &= \Theta(n \lg \lg n) \end{aligned}$$

c. $T(n) = 4T(n/2) + n^2 \sqrt{n}$

$$\Theta(n^2 \sqrt{n})$$

d. $T(n) = 3T(n/3 - 2) + n/2$

$$\Theta(n \lg n)$$

e. $T(n) = 2T(n/2) + n/\lg n$

Same as b,

$$\Theta(n \lg \lg n)$$

f. $T(n) = T(n/2) + T(n/4) + T(n/8) + n$

$$\Theta(n)$$

g. $T(n) = T(n - 1) + 1/n$

$$\begin{aligned} T(n) &= \sum_{i=1}^n \frac{1}{i} \\ &= \Theta(\lg n) \end{aligned}$$

h. $T(n) = T(n - 1) + \lg n$

$$\begin{aligned}
 T(n) &= \sum_{i=1}^n \lg i \\
 &= \lg n! \\
 &\leq \lg n^n \\
 &= \Theta(n \lg n)
 \end{aligned}$$

| $i.$ $T(n) = T(n - 2) + 1/\lg n$

$$\begin{aligned}
 T(n) &= \sum_{i=1}^{n/2} \frac{1}{\lg 2i} \\
 &= \Theta(\lg \lg n)
 \end{aligned}$$

| $j.$ $T(n) = \sqrt{n}T(\sqrt{n}) + n$

Let $n = 2^m$,

$$\begin{aligned}
 T(n) &= \sqrt{n}T(\sqrt{n}) + n \\
 T(2^m) &= 2^{m/2}T(2^{m/2}) + 2^m \\
 \frac{T(2^m)}{2^m} &= \frac{T(2^{m/2})}{2^{m/2}} + 1
 \end{aligned}$$

Let $S(m) = \frac{T(2^m)}{2^m}$, $S(m) = S(m/2) + 1 = \Theta(\lg m)$,

$$\therefore T(2^m) = \Theta(2^m \lg m)$$

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

4-4 Fibonacci numbers

This problem develops properties of the Fibonacci numbers, which are defined by recurrence (3.22). We shall use the technique of generating functions to solve the Fibonacci recurrence. Define the **generating function (or formal power series)** \mathcal{F} as

$$\begin{aligned}\mathcal{F}(z) &= \sum_{i=0}^{\infty} F_i z^i \\ &= 0 + z + z^2 + 2z^3 + 3z^4 + 5z^5 + 8z^6 + 13z^7 + 21z^8 + \dots\end{aligned}$$

where \mathcal{F}_i is the i th Fibonacci number.

a. Show that $\mathcal{F}(z) = z + z\mathcal{F}(z) + z^2\mathcal{F}(z)$.

$$\begin{aligned}z + z\mathcal{F}(z) + z^2\mathcal{F}(z) &= z + z \sum_{i=0}^{\infty} F_i z^i + z^2 \sum_{i=0}^{\infty} F_i z^i \\ &= z + \sum_{i=1}^{\infty} F_{i-1} z^i + \sum_{i=2}^{\infty} F_{i-2} z^i \\ &= z + z^2 + \sum_{i=2}^{\infty} (F_{i-1} + F_{i-2}) z^i \\ &= z + z^2 + \sum_{i=2}^{\infty} F_i z^i \\ &= \sum_{i=0}^{\infty} F_i z^i \\ &= \mathcal{F}(z)\end{aligned}$$

b. Show that

$$\begin{aligned}\mathcal{F}(z) &= \frac{z}{1 - z - z^2} \\ &= \frac{z}{(1 - \phi z)(1 - \hat{\phi} z)} \\ &= \frac{1}{\sqrt{5}} \left(\frac{1}{1 - \phi z} - \frac{1}{1 - \hat{\phi} z} \right)\end{aligned}$$

$$\phi = \frac{1+\sqrt{5}}{2} = 1.61803\dots$$

and

$$\hat{\phi} = \frac{1-\sqrt{5}}{2} = -0.61803\dots$$

$$\begin{aligned}\mathcal{F}(z) &= z + z\mathcal{F}(z) + z^2\mathcal{F}(z) \\ (1 - z - z^2)\mathcal{F}(z) &= z \\ \mathcal{F}(z) &= \frac{z}{1 - z - z^2}\end{aligned}$$

$$\begin{aligned}
 (1 - \phi z)(1 - \hat{\phi}z) &= 1 - (\phi + \hat{\phi})z + \phi\hat{\phi}z^2 \\
 \phi + \hat{\phi} &= 1 \\
 \phi\hat{\phi} &= \frac{1-5}{4} = -1 \\
 \therefore (1 - \phi z)(1 - \hat{\phi}z) &= 1 - z - z^2 \\
 \therefore \mathcal{F}(z) &= \frac{z}{(1 - \phi z)(1 - \hat{\phi}z)} \\
 \frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi z} - \frac{1}{1-\hat{\phi}z}\right) &= \frac{1}{\sqrt{5}}\left(\frac{(\hat{\phi}-\phi)z}{(1-\phi z)(1-\hat{\phi}z)}\right) \\
 \mathcal{F}(z) &= \frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi z} - \frac{1}{1-\hat{\phi}z}\right)
 \end{aligned}$$

c. Show that

$$\mathcal{F}(z) = \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i)z^i.$$

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x},$$

$$\begin{aligned}
 \mathcal{F}(z) &= \frac{1}{\sqrt{5}}\left(\frac{1}{1-\phi z} - \frac{1}{1-\hat{\phi}z}\right) \\
 &= \sum_{i=0}^{\infty} \frac{1}{\sqrt{5}}(\phi^i - \hat{\phi}^i)z^i
 \end{aligned}$$

d. Use part (c) to prove that $F_i = \phi^i/\sqrt{5}$ for $i > 0$, rounded to the nearest integer.
 (Hint: Observe that $|\hat{\phi}| < 1$.)

$$\frac{\hat{\phi}^i}{\sqrt{5}} \leq 0.5$$

4-5 Chip testing

Professor Diogenes has n supposedly identical integrated-circuit chips that in principle are capable of testing each other. The professor's test jig accommodates two chips at a time. When the jig is loaded, each chip tests the other and reports whether it is good or bad. A good chip always reports accurately whether the other chip is good or bad, but the professor cannot trust the answer of a bad chip. Thus, the four possible outcomes of a test are as follows:

Chip A says	Chip B says	Conclusion
B is good	A is good	both are good, or both are bad
B is good	A is bad	at least one is bad
B is bad	A is good	at least one is bad
B is bad	A is bad	at least one is bad

- a. Show that if more than $n/2$ chips are bad, the professor cannot necessarily determine which chips are good using any strategy based on this kind of pairwise test. Assume that the bad chips can conspire to fool the professor.

Symmetric.

- b. Consider the problem of finding a single good chip from among n chips, assuming that more than $n/2$ of the chips are good. Show that $\lfloor n/2 \rfloor$ pairwise tests are sufficient to reduce the problem to one of nearly half the size.

First assume n is even, then divide the chips in two groups, test each pair of chips with the same index from the two groups. If the result are is good, we keep one of chips; otherwise we remove both the chips. If n is odd, if there are odd number of chips left after the selections, then there must be more good chips than bad chips, we can simply discard the odd chip; otherwise if there are even number of chips, then if there are equal number of good and bad chips, the odd one must be good, and if there are more good chips than bad chips, the difference must be larger or equal to 2, therefore we can safely add the odd one to the set for next iteration.

$$T(n) = T(n/2) + n/2 = \sum_{i=0}^{\lg n-1} \frac{n}{2^i} \leq n/2$$

```

import random

class Chip:
    def __init__(self, state):
        self.state = state

    def check(self, other):
        if self.state:
            return other.state
        return random.randint(0, 1)

def check(chip_a, chip_b):
    return chip_a.check(chip_b) and chip_b.check(chip_a)

def choose_good_chip(chips):
    assert(len(chips) > 0)
    if len(chips) == 1:
        return chips[0]
    mid = len(chips) // 2
    next_chips = []
    for i in range(mid):
        if check(chips[i], chips[mid + i]):
            next_chips.append(chips[i])
    if len(chips) % 2 == 1 and len(next_chips) % 2 == 0:
        next_chips.append(chips[-1])
    return choose_good_chip(next_chips)

```

- c. Show that the good chips can be identified with $\Theta(n)$ pairwise tests, assuming that more than $n/2$ of the chips are good. Give and solve the recurrence that describes the number of tests.

Based on master method, $T(n) = T(n/2) + n/2 = \Theta(n)$

4-6 Monge arrays

An $m \times n$ array A of real numbers is a **Monge array** if for all i, j, k and l such that $1 \leq i < k \leq m$ and $1 \leq j < l \leq n$, we have

$$A[i, j] + A[k, l] \leq A[i, l] + A[k, j]$$

In other words, whenever we pick two rows and two columns of a Monge array and consider the four elements at the intersections of the rows and the columns, the sum of the upper-left and lower-right elements is less than or equal to the sum of the lower-left and upper-right elements. For example, the following array is Monge:

10	17	13	28	23
17	22	16	29	23
24	28	22	34	24
11	13	6	17	7
45	44	32	37	23
36	33	19	21	6
75	66	51	53	34

a. Prove that an array is Monge if and only if for all $i = 1, 2, \dots, m - 1$ and $j = 1, 2, \dots, n - 1$, we have

$$A[i, j] + A[i + 1, j + 1] \leq A[i, j + 1] + A[i + 1, j]$$

(Hint: For the "if" part, use induction separately on rows and columns.)

If $A[i, j] + A[i + 1, j + 1] \geq A[i, j + 1] + A[i + 1, j]$, it contradicts the definition of Monge arrays.

If $A[i, j] + A[i + 1, j + 1] \leq A[i, j + 1] + A[i + 1, j]$,

suppose $A[i, l - 1] + A[k - 1, l] \leq A[i, l] + A[k - 1, l - 1]$,

since $A[k - 1, l - 1] + A[k, l] \leq A[k - 1, l] + A[k, l - 1]$,

therefore $A[i, l - 1] + A[k, l] \leq A[i, l] + A[k, l - 1]$;

suppose $A[i, j] + A[k, l - 1] \leq A[i, l - 1] + A[k, j]$,

since $A[i, l - 1] + A[k, l] \leq A[i, l] + A[k, l - 1]$,

therefore $A[i, j] + A[k, l] \leq A[i, l] + A[k, j]$.

b. The following array is not Monge. Change one element in order to make it Monge.

(Hint: Use part (a).)

37	23	22	32
21	6	7	10
53	34	30	31
32	13	9	6
43	21	15	8

37	23	24	32
21	6	7	10
53	34	30	31
32	13	9	6
43	21	15	8

c. Let $f(i)$ be the index of the column containing the leftmost minimum element of row i . Prove that $f(1) \leq f(2) \leq \dots \leq f(m)$ for any $m \times n$ Monge array.

Let i and j be the index of leftmost minimal elements on row a and b , suppose $a < b$ and $i \geq j$.

$$A[a, i] \leq A[a, j],$$

$$A[b, j] \leq A[b, i],$$

$$A[a, j] + A[b, i] \leq A[a, i] + A[b, j],$$

the inequality is satisfied only when $i = j$, therefore $i \leq j$.

d. Here is a description of a divide-and-conquer algorithm that computes the leftmost minimum element in each row of an $m \times n$ Monge array A :

Construct a submatrix A' of A consisting of the even-numbered rows of A .

Recursively determine the leftmost minimum for each row of A' . Then compute the leftmost minimum in the odd-numbered rows of A .

Explain how to compute the leftmost minimum in the odd-numbered rows of A (given that the leftmost minimum of the even-numbered rows is known) in $O(m + n)$ time.

Search in the interval $[f(i - 1), f(i + 1)]$.

$$c_1 m/2 + c_2 n = O(m + n)$$

- e. Write the recurrence describing the running time of the algorithm described in part (d). Show that its solution is $O(m + n \log m)$.

$$\begin{aligned} T(m, n) &= T(m/2, n) + m + n \\ &= \sum_{i=0}^{\lg m - 1} \left(\frac{m}{2^i} + n \right) \\ &= \sum_{i=0}^{\lg m - 1} \left(\frac{m}{2^i} + n \right) \\ &= \frac{1}{1 - 1/2} m + n \lg m \\ &= 2m + n \lg m \\ &= O(m + n \log m) \end{aligned}$$

```
def get_min_index(arr):
    def get_min_index_rec(idx):
        if len(idx) == 1:
            min_idx = 0
            for j in range(1, len(arr[0])):
                if arr[idx[0]][j] < arr[idx[0]][min_idx]:
                    min_idx = j
            return [min_idx]
        sub_idx = [idx[i] for i in range(len(idx)) if i % 2 == 0]
        sub_min_idx = get_min_index_rec(sub_idx)
        sub_min_idx.append(len(arr[0]) - 1)
        min_idx = [sub_min_idx[i//2] for i in range(len(idx))]
        for i in range(1, len(idx), 2):
            for j in range(sub_min_idx[i//2] + 1, sub_min_idx[i//2 + 1] + 1):
                if arr[idx[i]][j] < arr[idx[i]][min_idx[i]]:
                    min_idx[i] = j
        return min_idx
    return get_min_index_rec([i for i in range(len(arr))])
```

5 Probabilistic Analysis and Randomized Algorithms

- 5.1 The hiring problem
- 5.2 Indicator random variables
- 5.3 Randomized algorithms
- 5.4 Probabilistic analysis and further uses of indicator random variables
- Problems

5.1 The hiring problem

5.1-1

Show that the assumption that we are always able to determine which candidate is best, in line 4 of procedure HIRE-ASSISTANT, implies that we know a total order on the ranks of the candidates.

Transitive

5.2-2 *

Describe an implementation of the procedure $\text{RANDOM } (a, b)$ that only makes calls to $\text{RANDOM } (0, 1)$. What is the expected running time of your procedure, as a function of a and b ?

Divide $[a, b]$ into $[a, mid]$ and $(mid, b]$, if $\text{RANDOM } (0, 1)$ gives 0 then we choose $[a, mid]$ and repeat the step until there is only one element left. The expected running time is $\Theta(\lg(b - a))$.

```
import random

def random_interval(a, b):
    while a < b:
        if random.randint(0, 1) == 0:
            b = (a + b) // 2
        else:
            a = (a + b) // 2 + 1
    return a
```

5.2-3 *

Suppose that you want to output 0 with probability $1/2$ and 1 with probability $1/2$. At your disposal is a procedure BIASED-RANDOM, that outputs either 0 or 1. It outputs 1 with some probability p and 0 with probability $1 - p$, where $0 < p < 1$, but you do not know what p is. Give an algorithm that uses BIASED-RANDOM as a subroutine, and returns an unbiased answer, returning 0 with probability $1/2$ and 1 with probability $1/2$. What is the expected running time of your algorithm as a function of p ?

The probabilities of generating $(0, 1)$ and $(1, 0)$ with BIASED-RANDOM is the same. We can generate two numbers with BIASED-RANDOM, and if they are different, we can return the first number, otherwise we should regenerate the two numbers. Since the probability of generating two different number is $2p(1 - p)$, thus the expectation of generation times is $\frac{1}{2p(1-p)}$.

```
import random

def biased_random():
    if random.random() < 0.1:
        return 0
    return 1

def unbiased_random():
    while True:
        a = biased_random()
        b = biased_random()
        if a != b:
            return a
```

5.2 Indicator random variables

5.2-1

In HIRE-ASSISTANT, assuming that the candidates are presented in a random order, what is the probability that you hire exactly one time? What is the probability that you hire exactly n times?

- Exactly one time

The best candidate comes first, which is $\frac{1}{n}$.

- Exactly n times

The candidates presented in ascending order, which is $\frac{1}{n!}$.

5.2-2

In HIRE-ASSISTANT, assuming that the candidates are presented in a random order, what is the probability that you hire exactly twice?

Suppose the first candidate is of rank k , followed by some candidates with rank less than k , then followed the candidate with rank n .

$$P = \sum_{k=1}^{n-1} \frac{1}{n} \frac{1}{n-k} = \frac{1}{n} \sum_{i=1}^{n-1} \frac{1}{i} = O\left(\frac{\lg n}{n}\right)$$

5.2-3

Use indicator random variables to compute the expected value of the sum of n dice.

$$E[X_i] = \frac{1+2+3+4+5+6}{6} = 3.5$$

$$\begin{aligned}
 E[X] &= E\left[\sum_{i=1}^n X_i\right] \\
 &= \sum_{i=1}^n E[X_i] \\
 &= \sum_{i=1}^n 3.5 \\
 &= 3.5n
 \end{aligned}$$

5.2-4

Use indicator random variables to solve the following problem, which is known as the **hat-check problem**. Each of n customers gives a hat to a hat-check person at a restaurant. The hat-check person gives the hats back to the customers in a random order. What is the expected number of customers who get back their own hat?

The probability of one customer get his/her hat back is $1/n$, therefore the expectation is $n(1/n) = 1$.

5.2-5

Let $A[1 \dots n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an **inversion** of A . (See Problem 2-4 for more on inversions.)

Suppose that the elements of A form a uniform random permutation of $\langle 1, 2, \dots, n \rangle$. Use indicator random variables to compute the expected number of inversions.

Suppose $X_{ij} = I\{(i, j) \text{ is an inversion}\}$,

$$E[X_{ij}] = \frac{1}{2}$$

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n E[X_{ij}] = \frac{n(n-1)}{4}$$

5.3 Randomized algorithms

5.3-1

Professor Marceau objects to the loop invariant used in the proof of Lemma 5.5. He questions whether it is true prior to the first iteration. He reasons that we could just as easily declare that an empty subarray contains no 0-permutations. Therefore, the probability that an empty subarray contains a 0-permutation should be 0, thus invalidating the loop invariant prior to the first iteration. Rewrite the procedure RANDOMIZE-IN-PLACE so that its associated loop invariant applies to a nonempty subarray prior to the first iteration, and modify the proof of Lemma 5.5 for your procedure.

...

5.3-2

Professor Kelp decides to write a procedure that produces at random any permutation besides the identity permutation. He proposes the following procedure:

```

PERMUTE-WITHOUT-IDENTITY(A)
1 n = A.length
2 for i = 1 to n - 1
3     swap A[i] with A[RANDOM(i + 1, n)]

```

Does this code do what Professor Kelp intends?

It's not uniform.

5.3-3

Suppose that instead of swapping element $A[i]$ with a random element from the subarray $A[i \dots n]$, we swapped it with a random element from anywhere in the array:

```

PERMUTE-WITH-ALL(A)
1 n = A.length
2 for i = 1 to n
3     swap A[i] with A[RANDOM(1, n)]

```

Does this code produce a uniform random permutation? Why or why not?

No. $n! \nmid n^n$.

5.3-4

Professor Armstrong suggests the following procedure for generating a uniform random permutation:

```

PERMUTE-BY-CYCPLIC(A)
1 n = A.length
2 let B[1..n] be a new array
3 offset = RANDOM(1, n)
4 for i = 1 to n
5     dest = i + offset
6     if dest > n
7         dest = dest - n
8     B[dest] = A[i]
9 return B

```

Show that each element $A[i]$ has a $1/n$ probability of winding up in any particular position in B . Then show that Professor Armstrong is mistaken by showing that the resulting permutation is not uniformly random.

$n! \nmid n^n$

5.3-5 *

Prove that in the array P in procedure PERMUTE-BY-SORTING, the probability that all elements are unique is at least $1 - 1/n$.

$$\begin{aligned}
P &= 1 \cdot \left(1 - \frac{1}{n^3}\right) \cdot \left(1 - \frac{2}{n^3}\right) \cdots \left(1 - \frac{n}{n^3}\right) \\
&\geq 1 \cdot \left(1 - \frac{n}{n^3}\right) \cdot \left(1 - \frac{n}{n^3}\right) \cdots \left(1 - \frac{n}{n^3}\right) \\
&\geq \left(1 - \frac{1}{n^2}\right)^n \\
&\geq 1 - n \cdot \frac{1}{n^2} \\
&= 1 - 1/n
\end{aligned}$$

5.3-6

Explain how to implement the algorithm PERMUTE-BY-SORTING to handle the case in which two or more priorities are identical. That is, your algorithm should produce a uniform random permutation, even if two or more priorities are identical.

Regenerate.

5.3-7

Suppose we want to create a **random sample** of the set $\{1, 2, 3, \dots, n\}$, that is, an m -element subset S , where $0 \leq m \leq n$, such that each m -subset is equally likely to be created. One way would be to set $A[i] = i$ for $i = 1, 2, 3, \dots, n$, call RANDOM-IN-PLACE(A), and then take just the first m array elements. This method would make n calls to the RANDOM procedure. If n is much larger than m , we can create a random sample with fewer calls to RANDOM. Show that the following recursive procedure returns a random m -subset S of $\{1, 2, 3, \dots, n\}$, in which each m -subset is equally likely, while making only m calls to RANDOM:

```
RANDOM-SAMPLE(m, n)
1 if m == 0
2     return \varnothing;
3 else S = RANDOM-SAMPLE(m - 1, n - 1)
4     i = RANDOM(1, n)
5     if i \in S
6         S = S \cup {n}
7     else S = S \cup {i}
8 return S
```

For $m = 1$, the subset is uniformly sampled with probability $1/n$.

Suppose RANDOM-SAMPLE $(m - 1, n - 1)$ creates an uniform subset,

for RANDOM-SAMPLE (m, n) , the probability of choosing n is:

$$\underbrace{\frac{n-1}{n}}_{i \in [1, n-1]} \cdot \underbrace{\frac{m-1}{n-1}}_{i \in S_{m-1}} + \underbrace{\frac{1}{n}}_{i=n} = \frac{m}{n}$$

the probability of k ($k < n$) is choosed is:

$$\underbrace{\frac{1}{n}}_{i=k} \cdot \underbrace{\frac{(n-1)-(m-1)}{n-1}}_{k \notin S_{m-1}} + \underbrace{\frac{m-1}{n-1}}_{k \in S_{m-1}} = \frac{m}{n}$$

5.4 Probabilistic analysis and further uses of indicator random variables

5.4-1

How many people must there be in a room before the probability that someone has the same birthday as you do is at least $1/2$? How many people must there be before the probability that at least two people have a birthday on July 4 is greater than $1/2$?

$$1 - \left(\frac{n-1}{n} \right)^m \geq \frac{1}{2}$$

$$m \geq \log_{364/365} \frac{1}{2} = 252.6519888441586$$

Thus there must be 253 people.

$$1 - \binom{m}{1} \frac{1}{n} \left(\frac{n-1}{n} \right)^{m-1} - \left(\frac{n-1}{n} \right)^m \geq \frac{1}{2}$$

There must be 613 people.

5.4-2

Suppose that we toss balls into b bins until some bin contains two balls. Each toss is independent, and each ball is equally likely to end up in any bin. What is the expected number of ball tosses?

Same as the birthday paradox.

5.4-3 *

For the analysis of the birthday paradox, is it important that the birthdays be mutually independent, or is pairwise independence sufficient? Justify your answer.

X_{ij} uses the pairwise independence, thus it is sufficient for proving.

5.4-4 *

How many people should be invited to a party in order to make it likely that there are *three* people with the same birthday?

$$\begin{aligned}
 X_{ijk} &= \frac{1}{n^2} \\
 E[X] &= E \left[\sum_{i=1}^m \sum_{j=i+1}^m \sum_{k=j+1}^m X_{ijk} \right] \\
 &= \sum_{i=1}^m \sum_{j=i+1}^m \sum_{k=j+1}^m E[X_{ijk}] \\
 &= \sum_{i=1}^m \sum_{j=i+1}^m \sum_{k=j+1}^m \frac{1}{n^2} \\
 &= \binom{m}{3} \frac{1}{n^2} \\
 &= \frac{m \cdot (m-1) \cdot (m-2)}{6n^2} \\
 m \cdot (m-1) \cdot (m-2) &\geq 6n^2
 \end{aligned}$$

At least 94 people.

5.4-5 *

What is the probability that a k -string over a set of size n forms a k -permutation?
How does this question relate to the birthday paradox?

Complementary to the birthday paradox.

$$Pr = 1 \cdot \frac{n-1}{n} \cdot \frac{n-2}{n} \cdot \dots \cdot \frac{n-k+1}{n}$$

5.4-6 *

Suppose that n balls are tossed into n bins, where each toss is independent and the ball is equally likely to end up in any bin. What is the expected number of empty bins?
What is the expected number of bins with exactly one ball?

- the expected number of empty bins

$$E[X_i] = \left(1 - \frac{1}{n}\right)^n \approx \frac{1}{e}$$

$$\begin{aligned} E[X] &= \sum_{i=1}^n E[X_i] \\ &= \sum_{i=1}^n \frac{1}{e} \\ &= \frac{n}{e} \end{aligned}$$

- the expected number of bins with exactly one ball

$$\begin{aligned} E[X_i] &= \binom{n}{1} \frac{1}{n} \left(1 - \frac{1}{n}\right)^{n-1} = \left(1 - \frac{1}{n}\right)^n \frac{n}{n-1} \approx \frac{1}{e} \\ E[X] &= \frac{n}{e} \end{aligned}$$

5.4-7 *

Sharpen the lower bound on streak length by showing that in n flips of a fair coin, the probability is less than $1/n$ that no streak longer than $\lg n - 2 \lg \lg n$ consecutive heads occurs.

...

Problems

5-1 Probabilistic counting

With a b -bit counter, we can ordinarily only count up to $2^b - 1$. With R. Morris's ***probabilistic counting***, we can count up to a much larger value at the expense of some loss of precision.

We let a counter value of i represent a count of n_i for $i = 0, 1, \dots, 2^b - 1$, where the n_i form an increasing sequence of nonnegative values. We assume that the initial value of the counter is 0, representing a count of $n_0 = 0$. The INCREMENT operation works on a counter containing the value i in a probabilistic manner. If $i = 2^b - 1$, then the operation reports an overflow error. Otherwise, the INCREMENT operation increases the counter by 1 with probability $1/(n_{i+1} - n_i)$, and it leaves the counter unchanged with probability $1 - 1/(n_{i+1} - n_i)$.

If we select $n_i = i$ for all $i \geq 0$, then the counter is an ordinary one. More interesting situations arise if we select, say, $n_i = 2^{i-1}$ for $i > 0$ or $n_i = F_i$ (the i th Fibonacci number - see Section 3.2).

For this problem, assume that n_{2^b-1} is large enough that the probability of an overflow error is negligible.

- a. Show that the expected value represented by the counter after n INCREMENT operations have been performed is exactly n .

$$E[X_i] = (n_{i+1} - n_i) \cdot \left(\frac{1}{n_{i+1} - n_i} \right) + 0 \cdot \left(1 - \frac{1}{n_{i+1} - n_i} \right) = 1$$

- b. The analysis of the variance of the count represented by the counter depends on the sequence of the n_i . Let us consider a simple case: $n_i = 100i$ for all $i \geq 0$. Estimate the variance in the value represented by the register after n INCREMENT operations have been performed.

$$\begin{aligned} Var[X_i] &= E[X_i^2] - E[X_i]^2 \\ &= \left(100^2 \cdot \frac{1}{100} + 0^2 \cdot \frac{99}{100} \right) - 1 \\ &= 99 \end{aligned}$$

$$Var[X] = \sum_{i=1}^n Var[X_i] = 99n$$

5-2 Searching an unsorted array

This problem examines three algorithms for searching for a value x in an unsorted array A consisting of n elements.

Consider the following randomized strategy: pick a random index i into A . If $A[i] = x$, then we terminate; otherwise, we continue the search by picking a new random index into A . We continue picking random indices into A until we find an index j such that $A[j] = x$ or until we have checked every element of A . Note that we pick from the whole set of indices each time, so that we may examine a given element more than once.

- a.** Write pseudocode for a procedure RANDOM-SEARCH to implement the strategy above. Be sure that your algorithm terminates when all indices into A have been picked.

```
import random

def random_search(a, x):
    n = len(a)
    searched = {}
    while len(searched) < n:
        r = random.randint(0, n - 1)
        if a[r] == x:
            return r
        if r not in searched.keys():
            searched[r] = True
    return -1
```

- b.** Suppose that there is exactly one index i such that $A[i] = x$. What is the expected number of indices into A that we must pick before we find x and RANDOM-SEARCH terminates?

n

- c.** Generalizing your solution to part (b), suppose that there are $k \geq 1$ indices i such that $A[i] = x$. What is the expected number of indices into A that we must pick before we find x and RANDOM-SEARCH terminates? Your answer should be a function of n and k .

n/k

- d. Suppose that there are no indices i such that $A[i] = x$. What is the expected number of indices into A that we must pick before we have checked all elements of A and RANDOM-SEARCH terminates?

Same as section 5.4.2, $n(\ln n + O(1))$

Now consider a deterministic linear search algorithm, which we refer to as DETERMINISTIC-SEARCH. Specifically, the algorithm searches A for x in order, considering $A[1], A[2], A[3], \dots, A[n]$ until either it finds $A[i] = x$ or it reaches the end of the array. Assume that all possible permutations of the input array are equally likely.

- e. Suppose that there is exactly one index i such that $A[i] = x$. What is the average-case running time of DETERMINISTIC-SEARCH? What is the worst-case running time of DETERMINISTIC_SEARCH?

Average: $n/2$

Worst: n

- f. Generalizing your solution to part (e), suppose that there are $k \geq 1$ indices i such that $A[i] = x$. What is the average-case running time of DETERMINISTIC-SEARCH? What is the worst-case running time of DETERMINISTIC-SEARCH? Your answer should be a function of n and k .

Average:

$$\begin{aligned} & \sum_{i=1}^n i \cdot \frac{\binom{n-i}{k-1}}{\binom{n}{k}} \\ &= \frac{n+1}{k+1} \end{aligned}$$

Worst: $n - k + 1$

- g. Suppose that there are no indices i such that $A[i] = x$. What is the average-case running time of DETERMINISTIC-SEARCH? What is the worst-case running time of DETERMINISTIC-SEARCH?

Average: n

Worst: n

Finally, consider a randomized algorithm SCRAMBLE-SEARCH that works by first randomly permuting the input array and then running the deterministic linear search given above on the resulting permuted array.

h. Letting k be the number of indices i such that $A[i] = x$, give the worst-case and expected running times of SCRAMBLE-SEARCH for the cases in which $k = 0$ and $k = 1$. Generalize your solution to handle the case in which $k \geq 1$.

Same as DETERMINISTIC-SEARCH.

i. Which of the three searching algorithms would you use? Explain your answer.

DETERMINISTIC-SEARCH

- Easy to implement
- $O(1)$ memory
- Guarantee $O(n)$ running time

6 Heapsort

- 6.1 Heaps
- 6.2 Maintaining the heap property
- 6.3 Building a heap
- 6.4 The heapsort algorithm
- 6.5 Priority queues
- Problems

6.1 Heaps

6.1-1

What are the minimum and maximum numbers of elements in a heap of height h ?

Minimum: $1 + 2 + 2^2 + \dots + 2^{h-1} + 1 = 2^h$

Maximum: $1 + 2 + 2^2 + \dots + 2^h = 2^{h+1} - 1$

6.1-2

Show that an n -element heap has height $\lfloor \lg n \rfloor$.

$$\begin{aligned} 2^h &\leq n & \leq 2^{h+1} - 1 \\ 2^h &\leq n & < 2^{h+1} \\ h &\leq \lg n & < h + 1 \end{aligned}$$

6.1-3

Show that in any subtree of a max-heap, the root of the subtree contains the largest value occurring anywhere in that subtree.

Transitivity of $A[i] \geq A[\text{LEFT}(i)], A[i] \geq A[\text{RIGHT}(i)]$

6.1-4

Where in a max-heap might the smallest element reside, assuming that all elements are distinct?

The leaves.

6.1-5

Is an array that is in sorted order a min-heap?

Yes, since $\text{PARENT}(i) < i, A[\text{PARENT}(i)] \leq A[i]$.

6.1-6

Is the array with values $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$ a max-heap?

No, $6 < 7$.

6.1-7

Show that, with the array representation for storing an n -element heap, the leaves are the nodes indexed by $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n$.

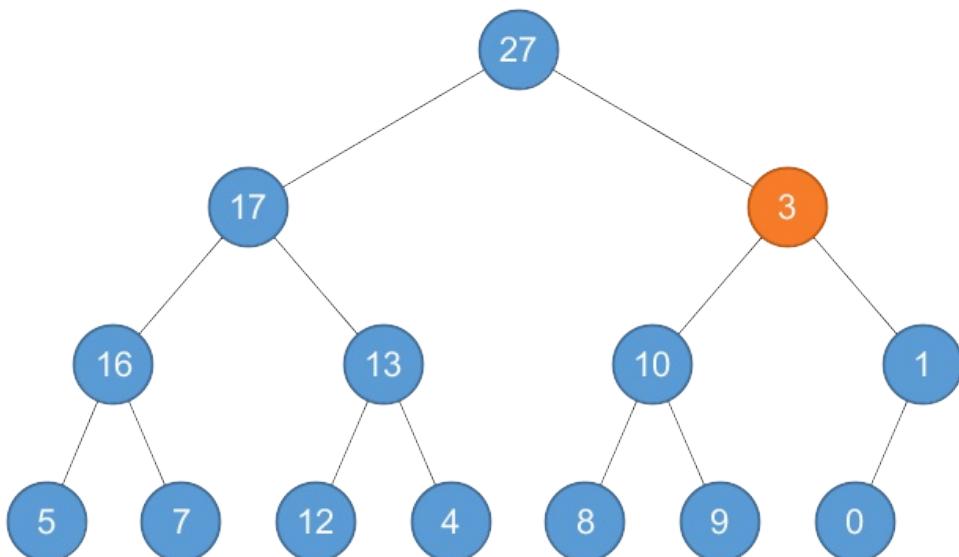
$$\begin{aligned}\text{LEFT}(i) &> n \\ 2i &> n \\ i &> n/2 \\ i &> \lfloor n/2 \rfloor + 1\end{aligned}$$

6.2 Maintaining the heap property

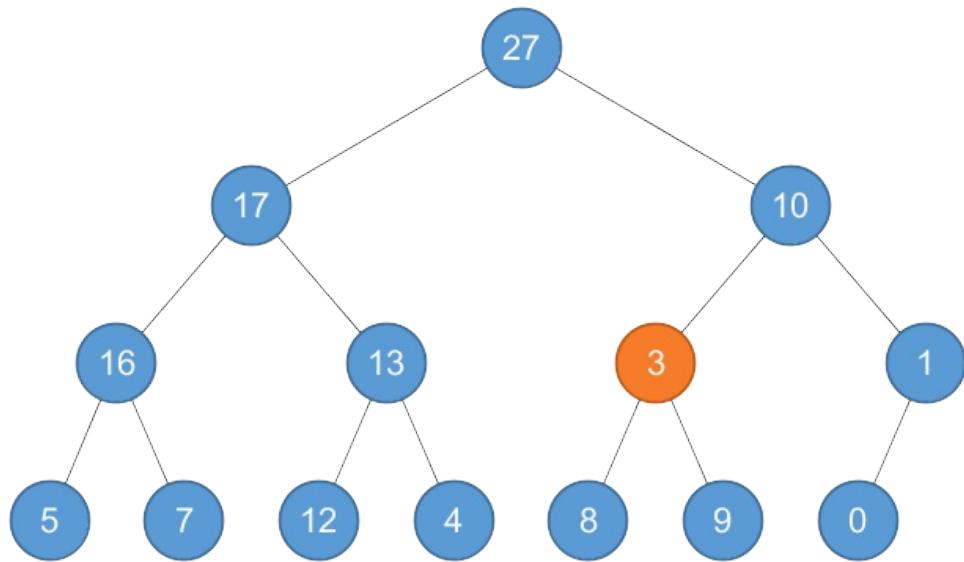
6.2-1

Using Figure 6.2 as a model, illustrate the operation of MAX-HEAPIFY $(A, 3)$ on the array $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$.

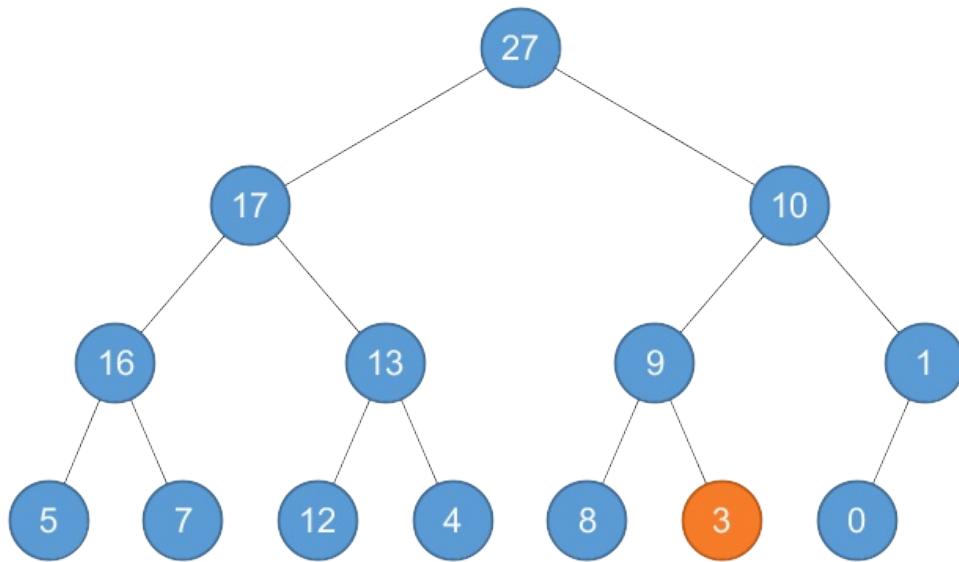
MAX-HEAPIFY $(A, 3)$, $A[3] = 3$, $A[6] = 10$, $A[7] = 1$, swap $A[3]$ and $A[6]$.



$$A = \langle 27, 17, 10, 16, 13, 3, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$$



MAX-HEAPIFY ($A, 6$) ; $A[6] = 3, A[12] = 8, A[13] = 9$, swap $A[6]$ and $A[13]$.



$$A = \langle 27, 17, 10, 16, 13, 9, 1, 5, 7, 12, 4, 8, 3, 0 \rangle$$

`MAX-HEAPIFY ($A, 13$) . $A[13] = 3$, done.`

6.2-2

Starting with the procedure MAX-HEAPIFY, write pseudocode for the procedure MIN-HEAPIFY (A, i) , which performs the corresponding manipulation on a minheap. How does the running time of MIN-HEAPIFY compare to that of MAX-HEAPIFY?

```

def parent(i):
    return (i - 1) >> 1

def left(i):
    return (i << 1) + 1

def right(i):
    return (i << 1) + 2

def min_heapify(a, i):
    min_idx = i
    l, r = left(i), right(i)
    if l < len(a) and a[l] < a[min_idx]:
        min_idx = l
    if r < len(a) and a[r] < a[min_idx]:
        min_idx = r
    if min_idx != i:
        a[i], a[min_idx] = a[min_idx], a[i]
        min_heapify(a, min_idx)

```

Running time is the same.

6.2-3

What is the effect of calling MAX-HEAPIFY ($A, 3$) when the element $A[i]$ is larger than its children?

No effect.

6.2-4

What is the effect of calling MAX-HEAPIFY ($A, 3$) for $i > A.\text{heap-size} = 2$?

No effect.

6.2-5

The code for MAX-HEAPIFY is quite efficient in terms of constant factors, except possibly for the recursive call in line 10, which might cause some compilers to produce inefficient code. Write an efficient MAX-HEAPIFY that uses an iterative control construct (a loop) instead of recursion.

```
def max_heapify(a, i):
    while True:
        max_idx = i
        l, r = left(i), right(i)
        if l < len(a) and a[l] > a[max_idx]:
            max_idx = l
        if r < len(a) and a[r] > a[max_idx]:
            max_idx = r
        if max_idx == i:
            break
        a[i], a[max_idx] = a[max_idx], a[i]
        i = max_idx
```

6.2-6

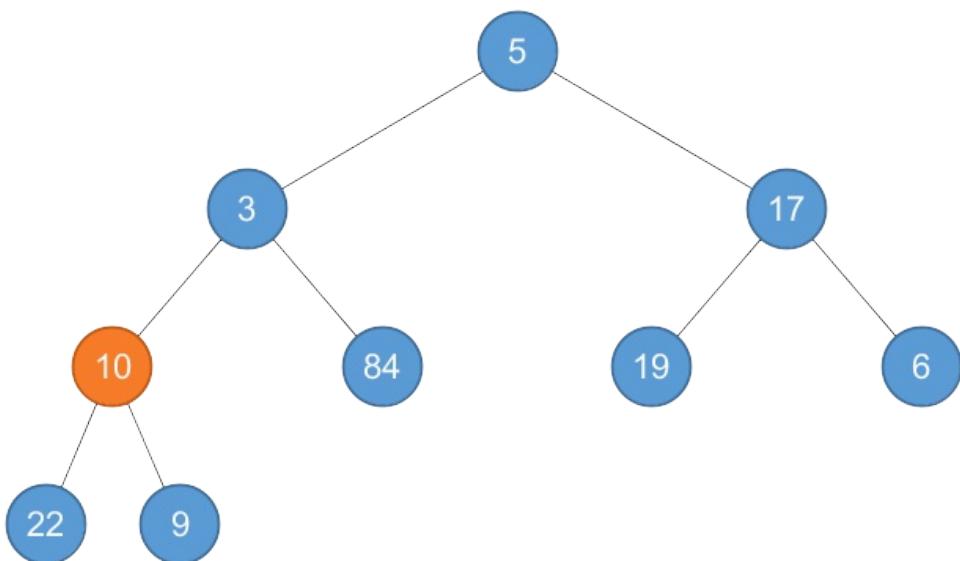
Show that the worst-case running time of MAX-HEAPIFY on a heap of size n is $\Omega(\lg n)$. (Hint: For a heap with n nodes, give node values that cause MAX-HEAPIFY to be called recursively at every node on a simple path from the root down to a leaf.)

The height is $\lfloor \lg n \rfloor$.

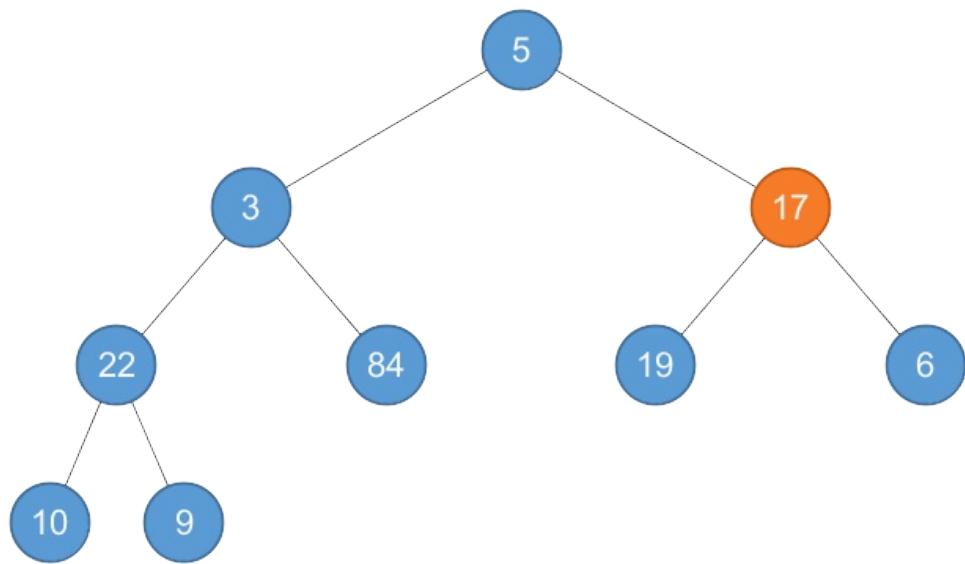
6.3 Building a heap

6.3-1

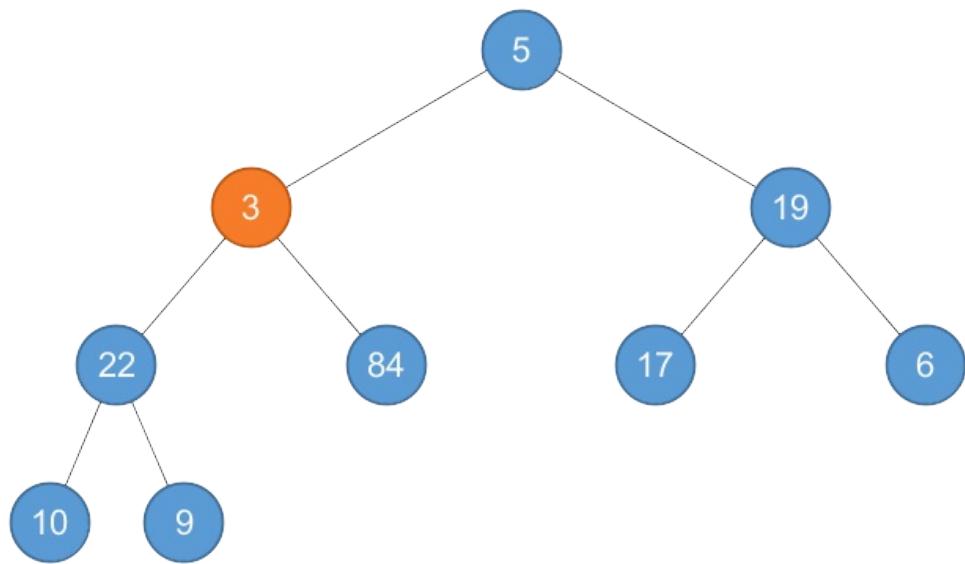
Using Figure 6.3 as a model, illustrate the operation of BUILD-MAX-HEAP on the array
 $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.



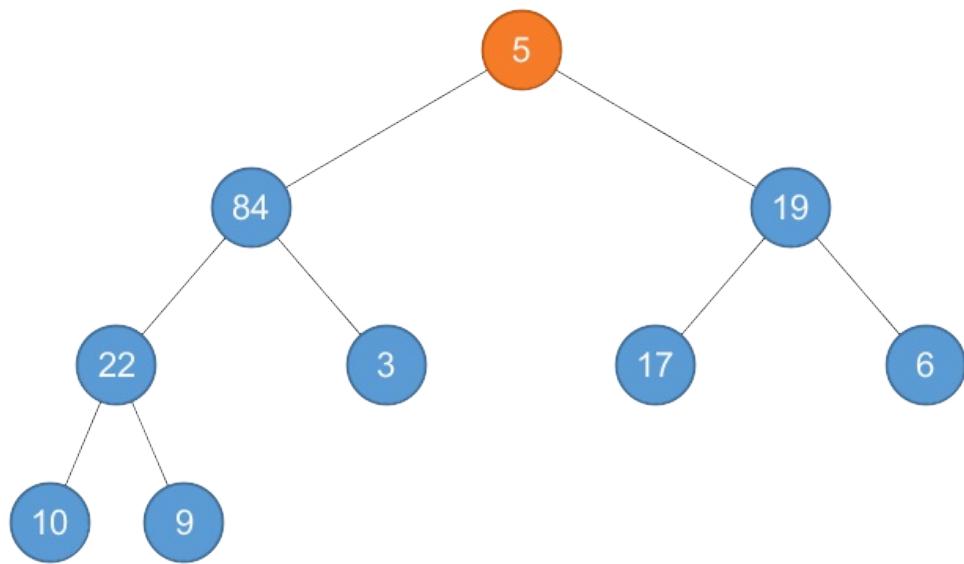
MAX-HEAPIFY ($A, 4$) ; $A = \langle 5, 3, 17, 22, 84, 19, 6, 10, 9 \rangle$



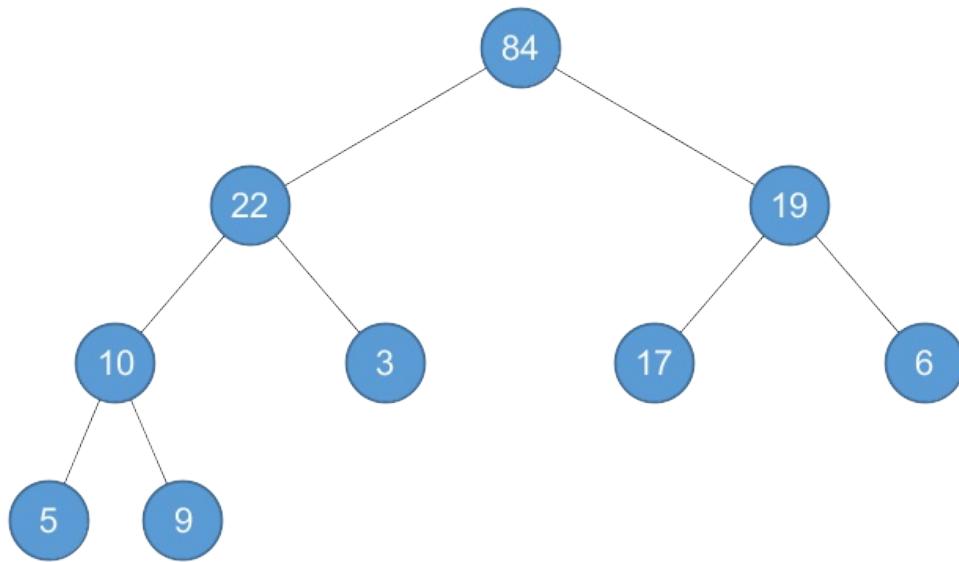
MAX-HEAPIFY ($A, 3$) ; $A = \langle 5, 3, 19, 22, 84, 17, 6, 10, 9 \rangle$



MAX-HEAPIFY ($A, 2$) : $A = \langle 5, 84, 19, 22, 3, 17, 6, 10, 9 \rangle$



MAX-HEAPIFY ($A, 1$) : $A = \langle 84, 22, 19, 10, 3, 17, 6, 5, 9 \rangle$

**6.3-2**

Why do we want the loop index i in line 2 of BUILD-MAX-HEAP to decrease from $\lfloor A.length/2 \rfloor$ to 1 rather than increase from 1 to $\lfloor A.length/2 \rfloor$?

To ensure the subtrees are heaps.

6.3-3

Show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n -element heap.

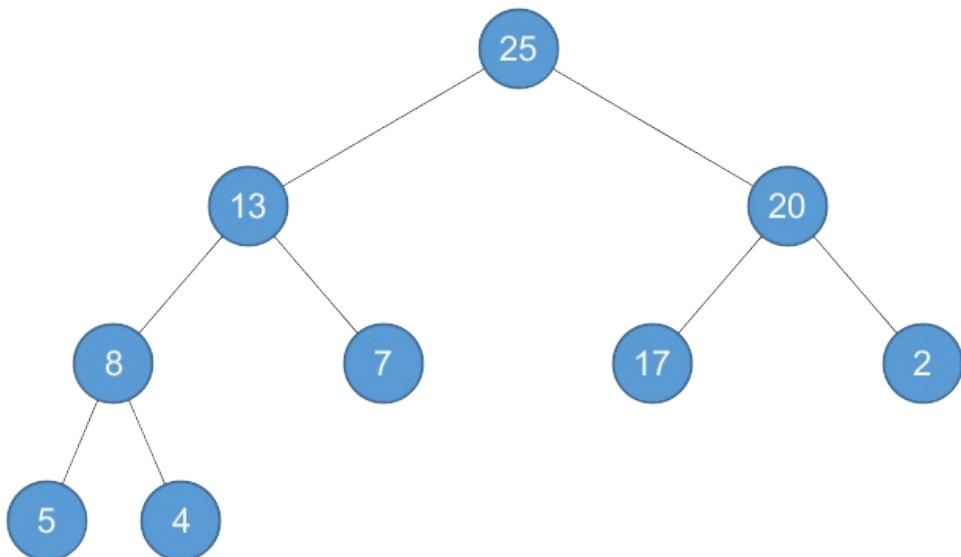
$$\frac{n - \lceil n/2 \rceil}{2^h} = \left\lceil \frac{n}{2^{h+1}} \right\rceil$$

6.4 The heapsort algorithm

6.4-1

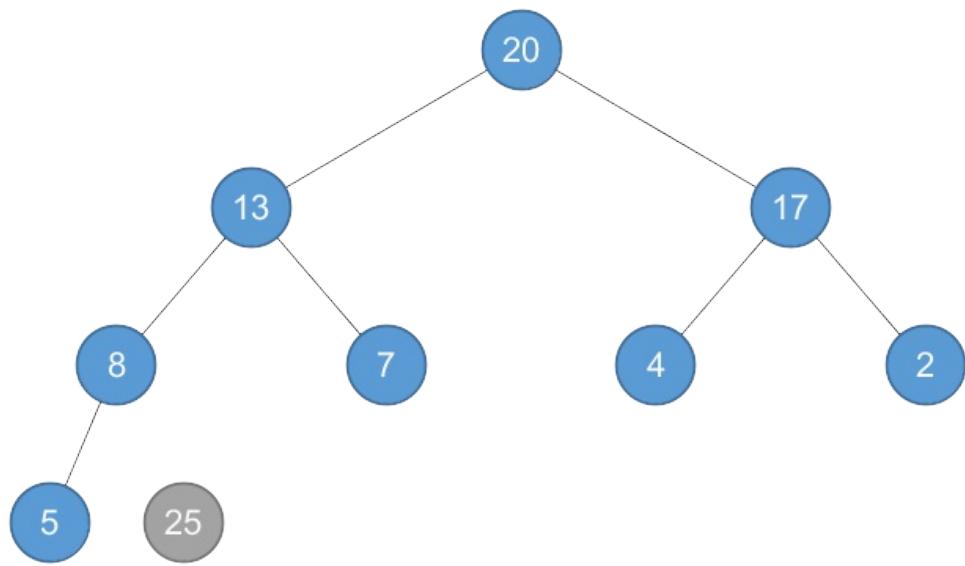
Using Figure 6.4 as a model, illustrate the operation of HEAPSORT on the array
 $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$

$\text{BUILD_MAX_HEAP}(A) : A = \langle 25, 13, 20, 8, 7, 17, 2, 5, 4 \rangle$

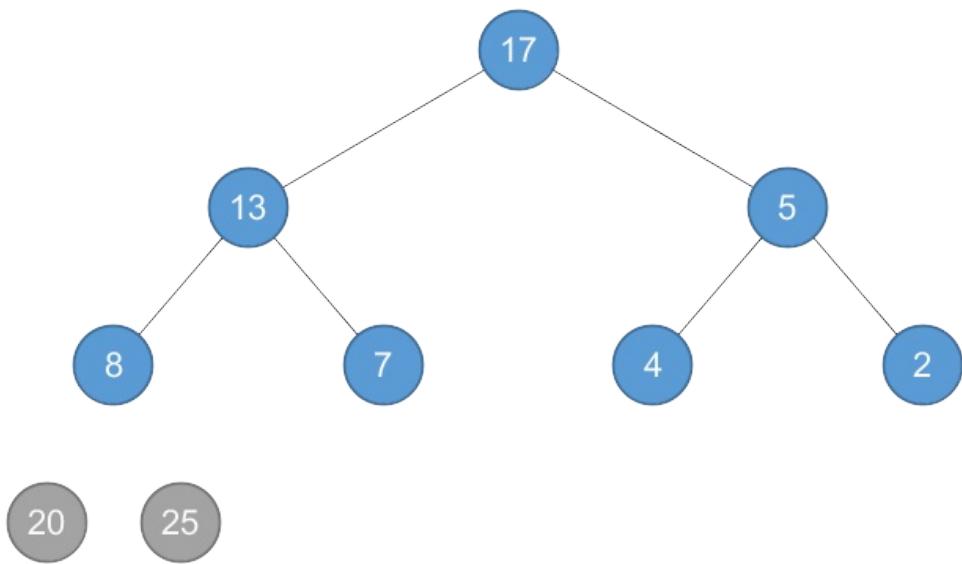


$\text{HEAPSORT}(A) :$

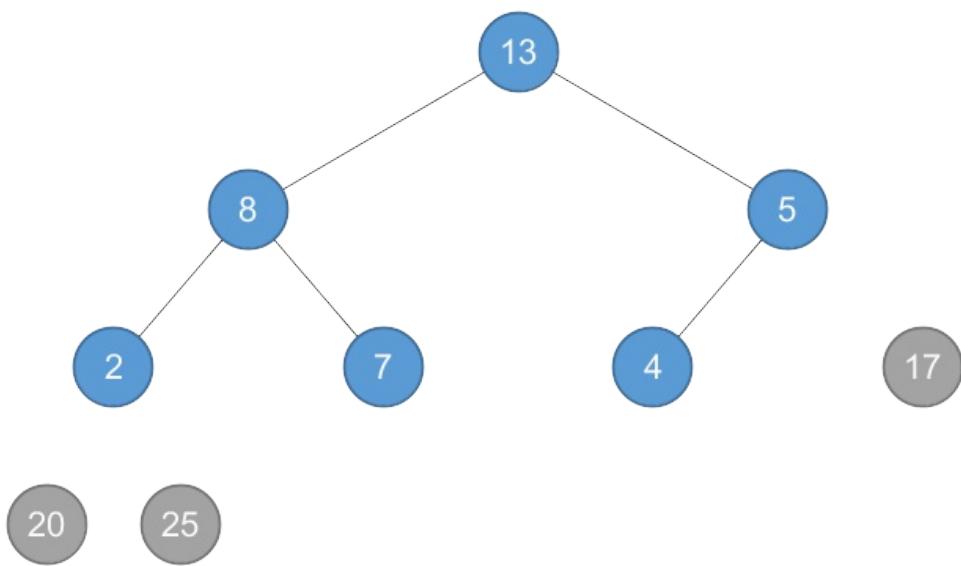
$A = \langle 20, 13, 17, 8, 7, 4, 2, 5, |25\rangle$



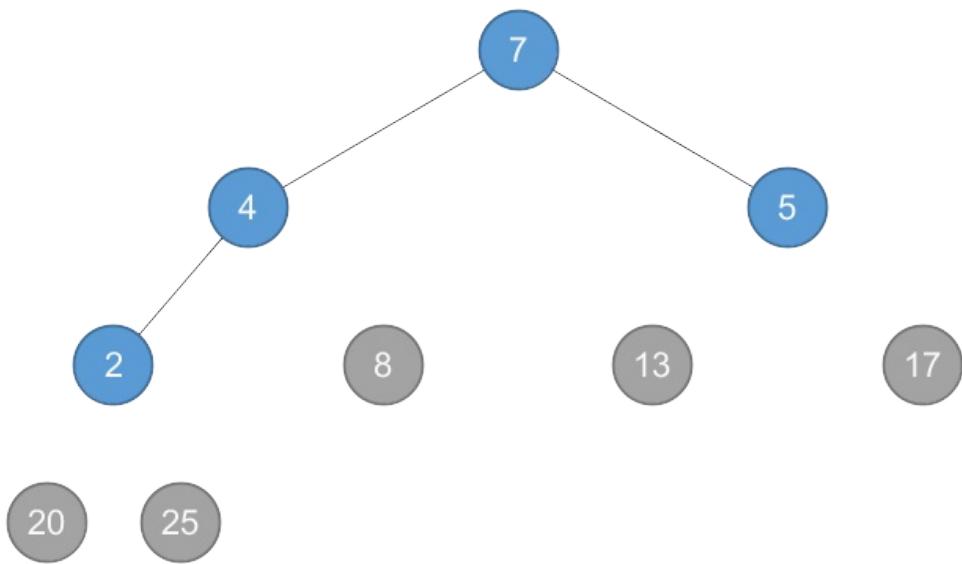
$$A = \langle 17, 13, 5, 8, 7, 4, 2, | 20, 25 \rangle$$



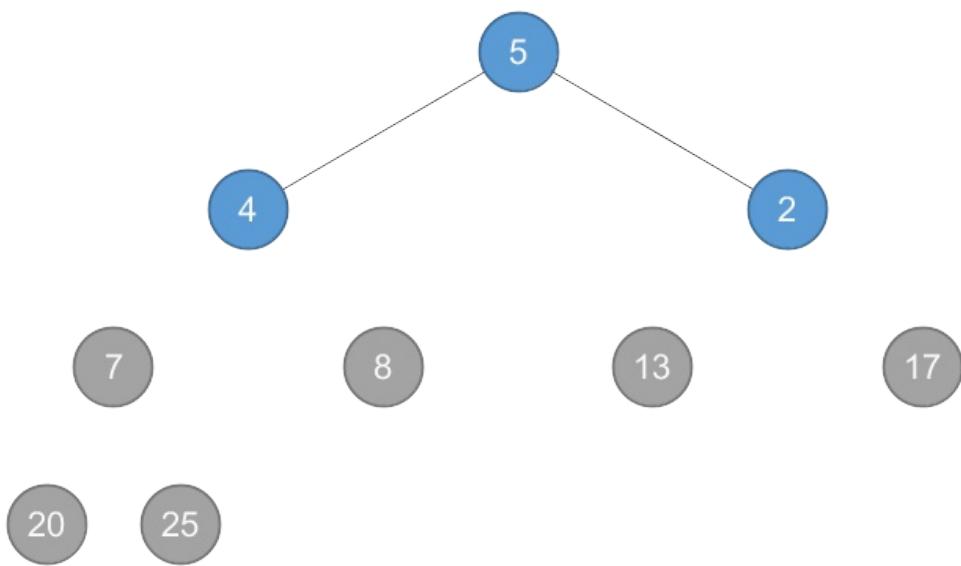
$$A = \langle 13, 8, 5, 2, 7, 4, | 17, 20, 25 \rangle$$



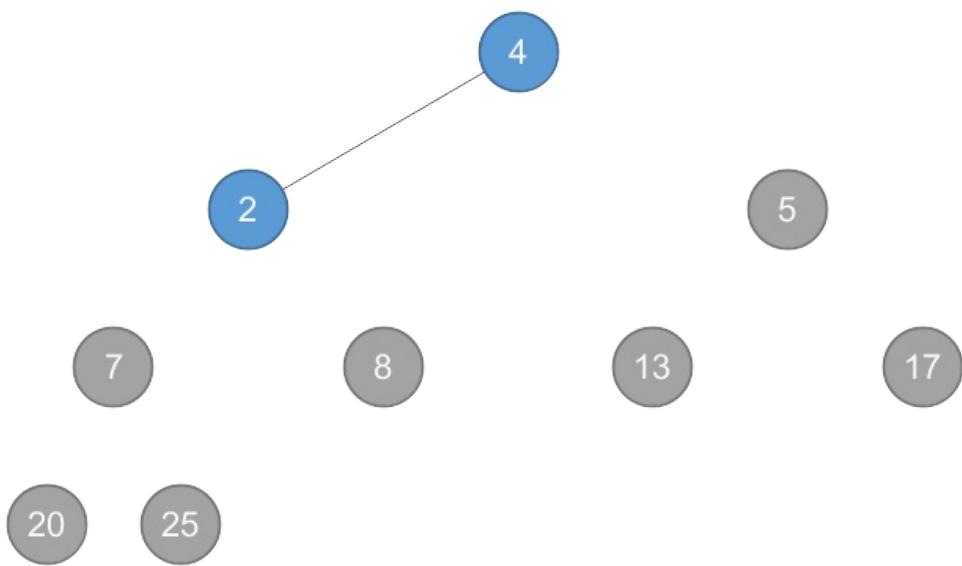
$$A = \langle 8, 7, 5, 2, 4, | 13, 17, 20, 25 \rangle$$



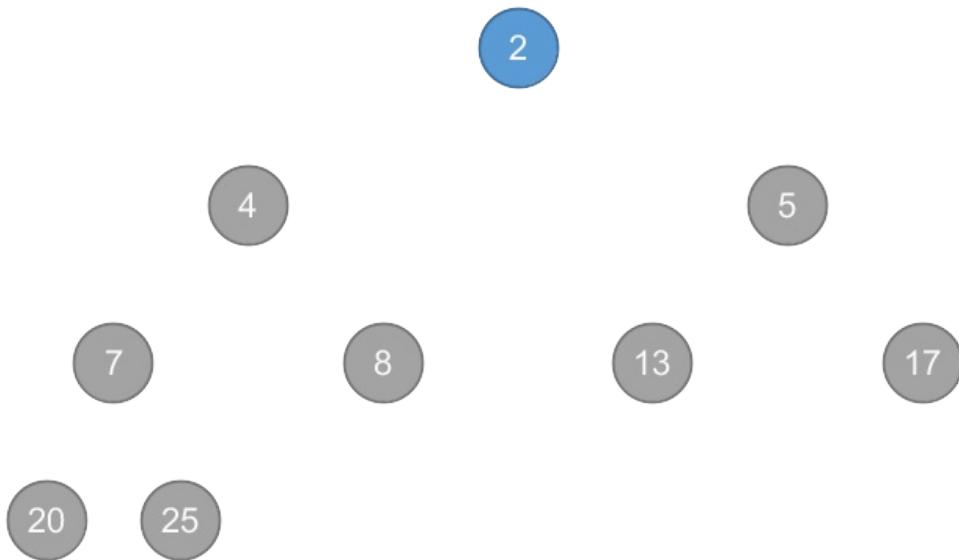
$$A = \langle 7, 4, 5, 2, | 8, 13, 17, 20, 25 \rangle$$



$$A = \langle 5, 4, 2, | 7, 8, 13, 17, 20, 25 \rangle$$



$$A = \langle 4, 2, | 5, 7, 8, 13, 17, 20, 25 \rangle$$



$$A = \langle 2, | 4, 5, 7, 8, 13, 17, 20, 25 \rangle$$

6.4-2

Argue the correctness of HEAPSORT using the following loop invariant:

At the start of each iteration of the for loop of lines 2–5, the subarray $A[1 \dots i]$ is a max-heap containing the i smallest elements of $A[1 \dots n]$, and the subarray $A[i + 1 \dots n]$ contains the $n - i$ largest elements of $A[1 \dots n]$, sorted.

In each iteration we move the largest element to the sorted array.

6.4-3

What is the running time of HEAPSORT on an array A of length n that is already sorted in increasing order? What about decreasing order?

Both are $\Theta(n \lg n)$ since there are n calls to MAX-HEAPIFY.

6.4-4

Show that the worst-case running time of HEAPSORT is $\Omega(n \lg n)$.

BUILD-HEAD is $\Theta(n)$ and MAX-HEAPIFY is $\Theta(n \lg n)$.

6.4-5 *

Show that when all elements are distinct, the best-case running time of HEAPSORT is $\Omega(n \lg n)$.

...

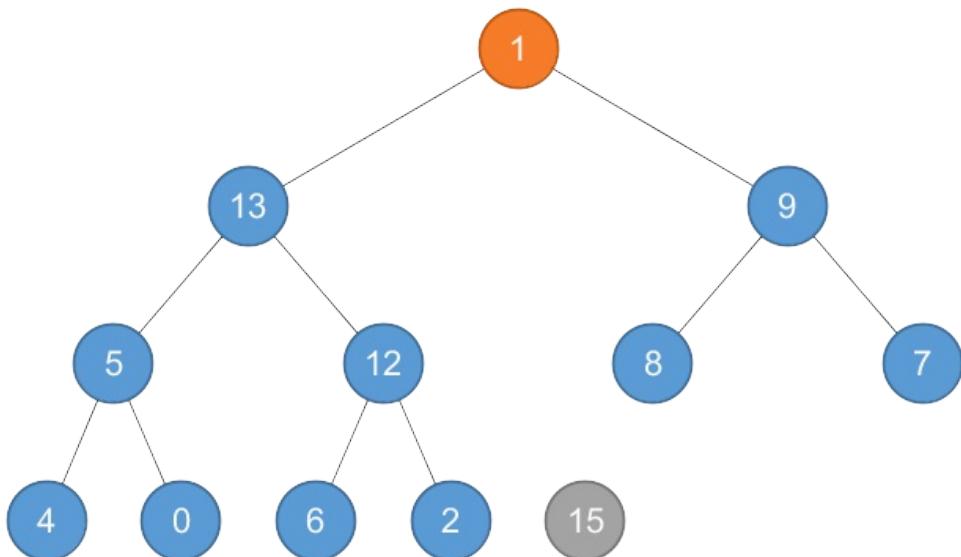
6.5 Priority queues

6.5-1

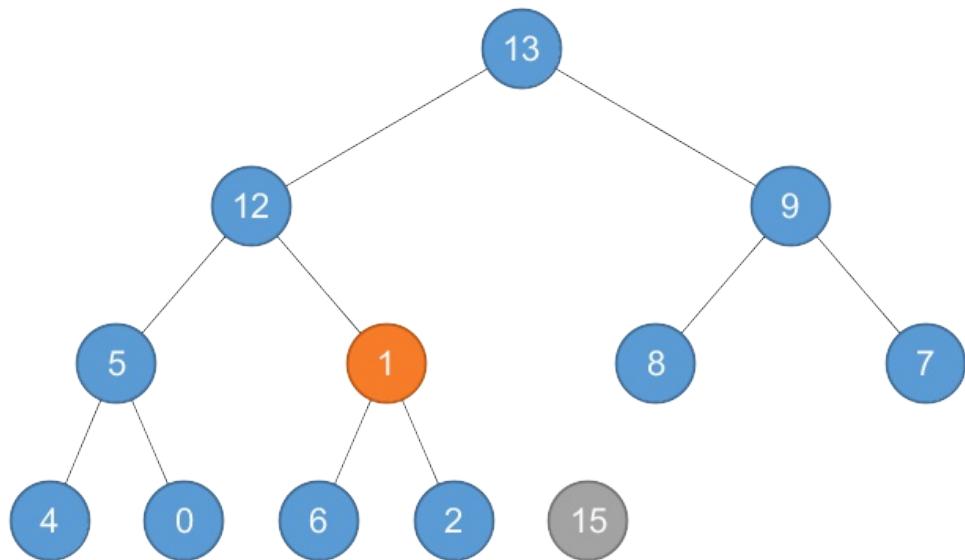
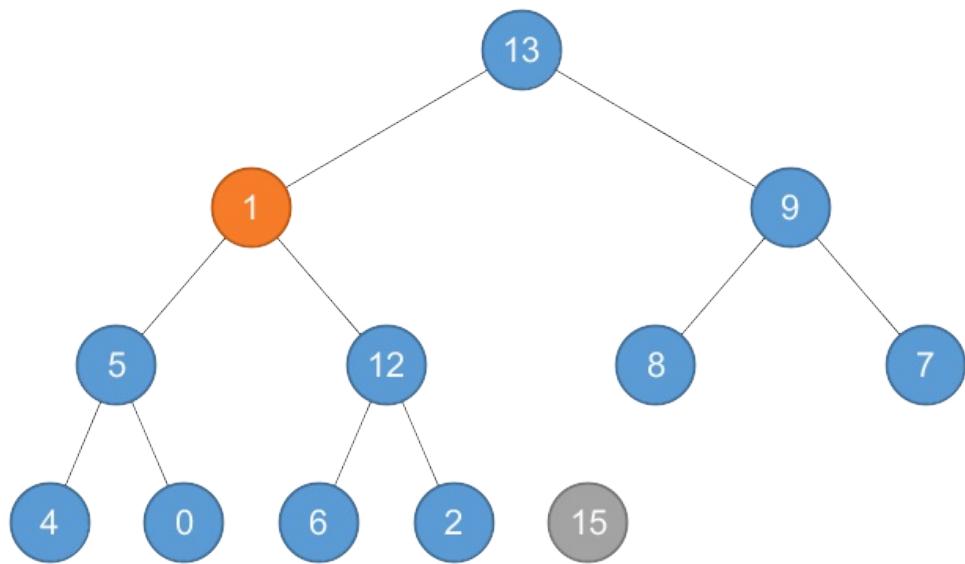
Illustrate the operation of HEAP-EXTRACT-MAX on the heap

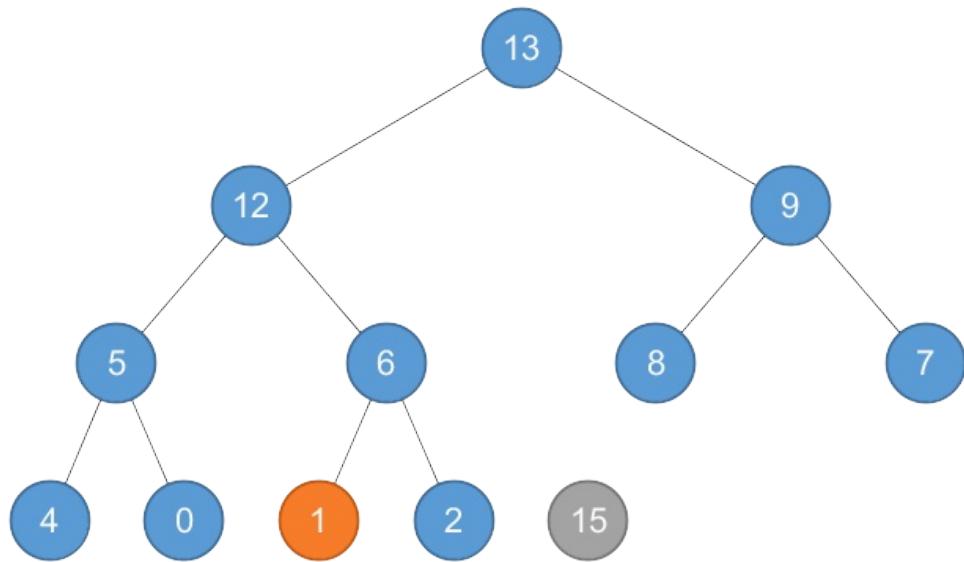
$$A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$$

Return 15 and $A = \langle 1, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2 \rangle$,



MAX-HEAPIFY (A) ; $A = \langle 13, 12, 9, 5, 6, 8, 7, 4, 0, 1, 2 \rangle$

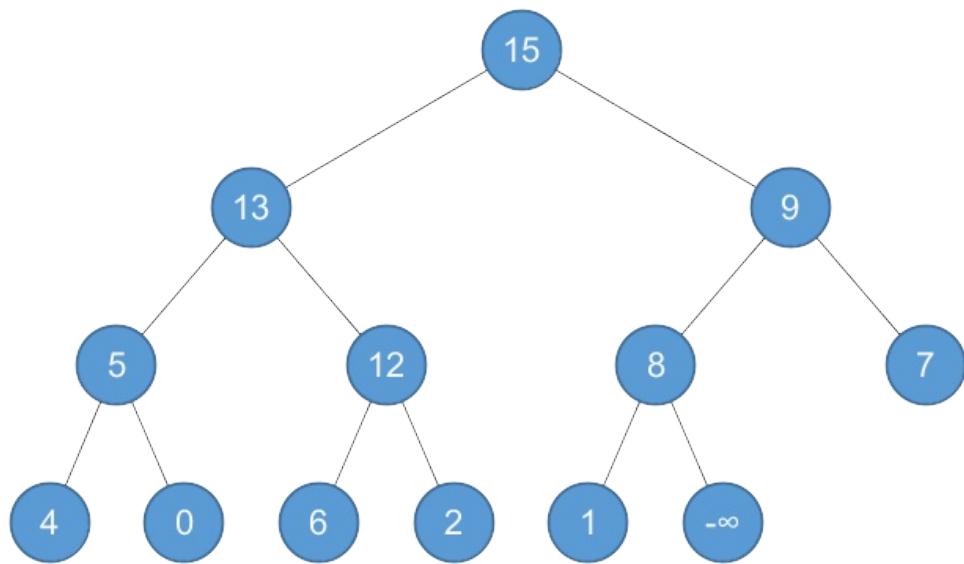




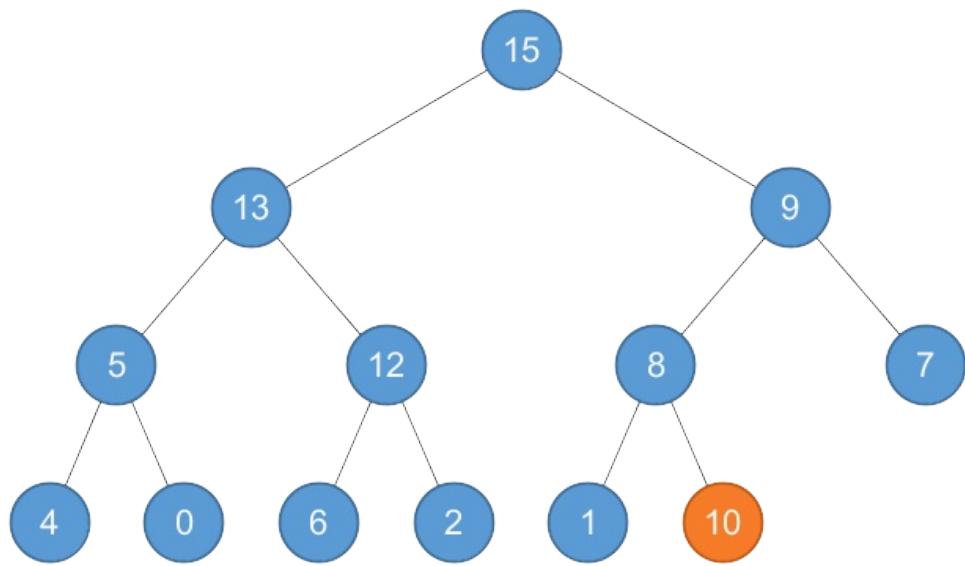
6.5-2

Illustrate the operation of MAX-HEAP-INSERT ($A, 10$) on the heap
 $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 \rangle$

Insert: $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1, -\infty \rangle$

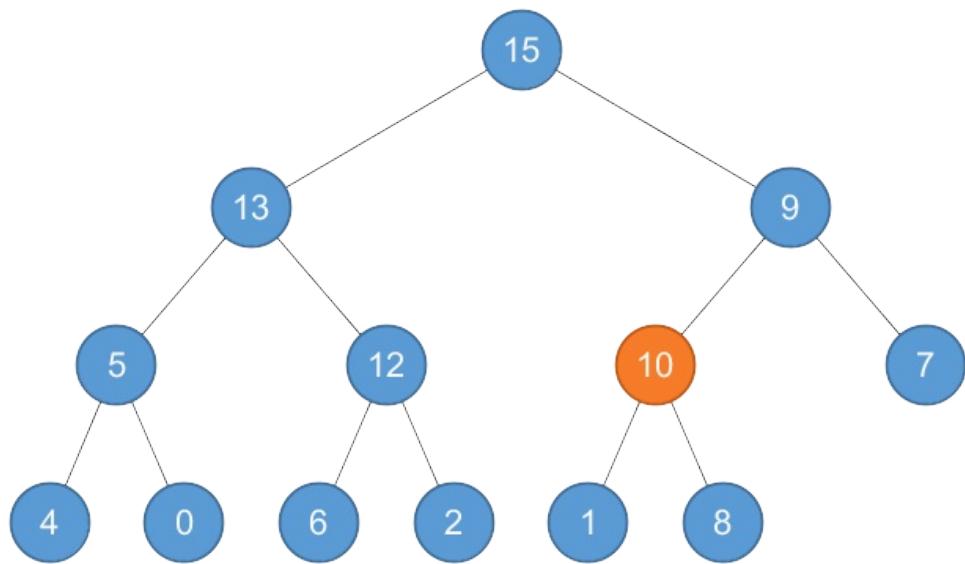


Increase: $A = \langle 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1, 10 \rangle$

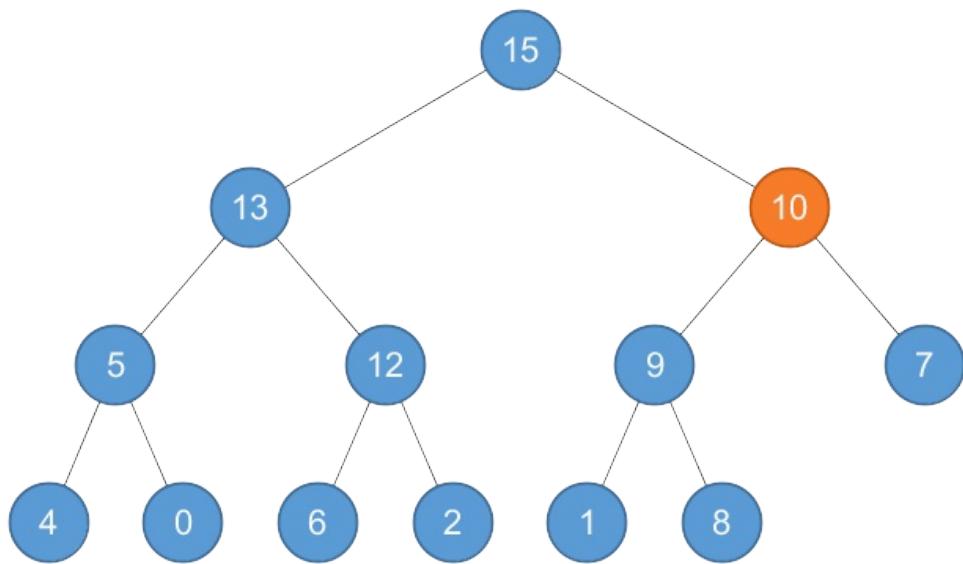


Heapify:

$$A = \langle 15, 13, 9, 5, 12, 10, 7, 4, 0, 6, 2, 1, 8 \rangle$$



$$A = \langle 15, 13, 10, 5, 12, 9, 7, 4, 0, 6, 2, 1, 8 \rangle$$



6.5-3

Write pseudocode for the procedures HEAP-MINIMUM, HEAP-EXTRACT-MIN, HEAP-DECREASE-KEY, and MIN-HEAP-INSERT that implement a min-priority queue with a min-heap.

```

def heap_minimum(a):
    assert(len(a) > 0)
    return a[0]

def heap_extract_min(a):
    assert(len(a) > 0)
    val = a[0]
    a[0] = a[-1]
    del a[-1]
    min_heapify(a, 0)
    return val

def heap_decrease_key(a, i, key):
    assert(key <= a[i])
    a[i] = key
    while i > 0 and a[i] < a[parent(i)]:
        a[i], a[parent(i)] = a[parent(i)], a[i]
        i = parent(i)

def min_heap_insert(a, key):
    a.append(1e100)
    heap_decrease_key(a, len(a) - 1, key)

```

6.5-4

Why do we bother setting the key of the inserted node to $-\infty$ in line 2 of MAXHEAP-INSERT when the next thing we do is increase its key to the desired value?

To make $key \geq A[i]$.

6.5-5

Argue the correctness of HEAP-INCREASE-KEY using the following loop invariant:

At the start of each iteration of the while loop of lines 4–6, the subarray $A[1 \dots A.\text{heap-size}]$ satisfies the max-heap property, except that there may be one violation: $A[i]$ may be larger than $A[\text{PARENT}(i)]$.

You may assume that the subarray $A[1 \dots A.\text{heap-size}]$ satisfies the max-heap property at the time HEAP-INCREASE-KEY is called.

Correct.

6.5-6

Each exchange operation on line 5 of HEAP-INCREASE-KEY typically requires three assignments. Show how to use the idea of the inner loop of INSERTION-SORT to reduce the three assignments down to just one assignment.

```
def heap_increase_key(a, i, key):
    assert(key >= a[i])
    while i > 0 and key > a[parent(i)]:
        a[i] = a[parent(i)]
        i = parent(i)
    a[i] = key
```

6.5-7

Show how to implement a first-in, first-out queue with a priority queue. Show how to implement a stack with a priority queue.

```
class Queue:
    def __init__(self):
        self.heap = []
        self.inc = 0

    def push(self, val):
        self.inc += 1
        min_heap_insert(self.heap, (self.inc, val))

    def front(self):
        return heap_minimum(self.heap)

    def pop(self):
        return heap_extract_min(self.heap)

class Stack:
    def __init__(self):
        self.heap = []
        self.inc = 0

    def push(self, val):
        self.inc += 1
        max_heap_insert(self.heap, (self.inc, val))

    def top(self):
        return heap_maximum(self.heap)

    def pop(self):
        return heap_extract_max(self.heap)
```

6.5-8

The operation $\text{HEAP-DELETE}(A, i)$ deletes the item in node i from heap A . Give an implementation of HEAP-DELETE that runs in $O(\lg n)$ time for an n -element max-heap.

```
def heap_delete(a, i):
    if i == len(a) - 1:
        del a[-1]
    else:
        a[i] = a[-1]
        del a[-1]
        max_heapify(a, i)
        heap_increase_key(a, i, a[i])
```

6.5-9

Give an $O(n \lg k)$ -time algorithm to merge k sorted lists into one sorted list, where n is the total number of elements in all the input lists. (Hint: Use a minheap for k -way merging.)

```
def merge_lists(lists):
    k = len(lists)
    heap = []
    for i in range(k):
        if len(lists[i]) > 0:
            min_heap_insert(heap, (lists[i][0], i))
    idx = [0 for lst in lists]
    a = []
    while len(heap) > 0:
        val, i = heap_extract_min(heap)
        a.append(val)
        idx[i] += 1
        if idx[i] < len(lists[i]):
            min_heap_insert(heap, (lists[i][idx[i]], i))
    return a
```

Problems

6-1 Building a heap using insertion

We can build a heap by repeatedly calling MAX-HEAP-INSERT to insert the elements into the heap. Consider the following variation on the BUILD-MAX-HEAP procedure:

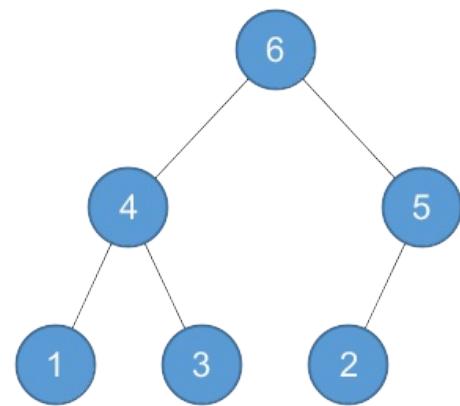
```
BUILD-MAX-HEAP' (A)
1 A.heap-size = 1
2 for i = 2 to A.length
3     MAX-HEAP-INSERT(A, A[i])
```

- a. Do the procedures BUILD-MAX-HEAP and BUILD-MAX-HEAP' always create the same heap when run on the same input array? Prove that they do, or provide a counterexample.

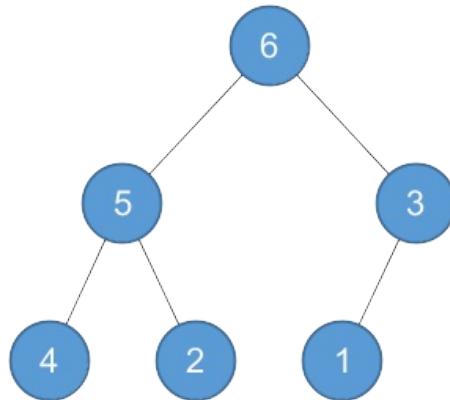
No.

For $\langle 1, 2, 3, 4, 5, 6 \rangle$,

BUILD-MAX-HEAP: $\langle 6, 4, 5, 1, 3, 2 \rangle$;



BUILD-MAX-HEAP: $\langle 6, 5, 3, 4, 2, 1 \rangle$



- b.** Show that in the worst case, BUILD-MAX-HEAP' requires $\Theta(n \lg n)$ time to build an n -element heap.

MAX-HEAP-INSERT is $\Theta(\lg n)$, thus BUILD-MAX-HEAP' is $\Theta(n \lg n)$.

6-2 Analysis of d -ary heaps

A d -ary heap is like a binary heap, but (with one possible exception) non-leaf nodes have d children instead of 2 children.

- a.** How would you represent a d -ary heap in an array?

If the index of the array begins with 0, then the k th children of node i is $id + k$. The

parent of node i is $\left\lfloor \frac{i - 1}{d} \right\rfloor$.

Thus if the index begins with 1, the k th children is $(i - 1)d + k + 1$, the parent is

$$\left\lfloor \frac{i - 2}{d} \right\rfloor + 1$$

b. What is the height of a d -ary heap of n elements in terms of n and d ?

$$\log_d n$$

c. Give an efficient implementation of EXTRACT-MAX in a d -ary max-heap. Analyze its running time in terms of d and n .

$$\Theta(d \log_d n)$$

```

def parent(d, i):
    return (i - 1) // d

def child(d, i, k):
    return (i * d) + k

def max_heapify(d, a, i):
    max_idx = i
    for k in range(1, d + 1):
        c = child(d, i, k)
        if c < len(a) and a[c] > a[max_idx]:
            max_idx = c
    if max_idx != i:
        a[i], a[max_idx] = a[max_idx], a[i]
        max_heapify(d, a, max_idx)

def extract_max(d, a):
    assert(len(a) > 0)
    val = a[0]
    a[0] = a[-1]
    del a[-1]
    max_heapify(d, a, 0)
    return val

```

d. Give an efficient implementation of INSERT in a d -ary max-heap. Analyze its running time in terms of d and n .

$$\Theta(\log_d n)$$

```

def increase_key(d, a, i, key):
    assert(key >= a[i])
    while i > 0 and key > a[parent(d, i)]:
        a[i] = a[parent(d, i)]
        i = parent(d, i)
    a[i] = key

def insert(d, a, key):
    a.append(-1e100)
    increase_key(d, a, len(a) - 1, key)

```

- e. Give an efficient implementation of INCREASE-KEY (A, i, k) , which flags an error if $k < A[i]$, but otherwise sets $A[i] = k$ and then updates the d -ary maxheap structure appropriately. Analyze its running time in terms of d and n .

 $\Theta(\log_d n)$

```

def increase_key(d, a, i, key):
    assert(key >= a[i])
    while i > 0 and key > a[parent(d, i)]:
        a[i] = a[parent(d, i)]
        i = parent(d, i)
    a[i] = key

```

6-3 Young tableaus

An $m \times n$ **Young tableau** is an $m \times n$ matrix such that the entries of each row are in sorted order from left to right and the entries of each column are in sorted order from top to bottom. Some of the entries of a Young tableau may be ∞ , which we treat as nonexistent elements. Thus, a Young tableau can be used to hold $r \leq mn$ finite numbers.

- a. Draw a 4×4 Young tableau containing the elements $\{9, 16, 3, 2, 4, 8, 5, 14, 12\}$.

	2	3	4	5
	8	9	12	14
	16	∞	∞	∞
	∞	∞	∞	∞

- b.** Argue that an $m \times n$ Young tableau Y is empty if $Y[1, 1] = \infty$. Argue that Y is full (contains mn elements) if $Y[m, n] < \infty$.

Transitive.

- c.** Give an algorithm to implement EXTRACT-MIN on a nonempty $m \times n$ Young tableau that runs in $O(m + n)$ time. Your algorithm should use a recursive subroutine that solves an $m \times n$ problem by recursively solving either an $(m - 1) \times n$ or an $m \times (n - 1)$ subproblem. Define $T(p)$, where $p = m + n$, to be the maximum running time of EXTRACT-MIN on any $m \times n$ Young tableau. Give and solve a recurrence for $T(p)$ that yields the $O(m + n)$ time bound.

```
def extract_min(a):
    m, n = len(a), len(a[0])
    val = a[0][0]
    a[0][0] = 1e8

    def maintain(i, j):
        min_i, min_j = i, j
        if i + 1 < m and a[i + 1][j] < a[min_i][min_j]:
            min_i, min_j = i + 1, j
        if j + 1 < n and a[i][j + 1] < a[min_i][min_j]:
            min_i, min_j = i, j + 1
        if min_i != i or min_j != j:
            a[i][j], a[min_i][min_j] = a[min_i][min_j], a[i][j]
            maintain(min_i, min_j)

    maintain(0, 0)
    return val
```

$$T(n) = T(n - 1) + O(1)$$

- d.** Show how to insert a new element into a nonfull $m \times n$ Young tableau in $O(m + n)$ time.

```

def insert(a, val):
    m, n = len(a), len(a[0])
    a[m - 1][n - 1] = val

def maintain(i, j):
    max_i, max_j = i, j
    if i - 1 >= 0 and a[i - 1][j] > a[max_i][max_j]:
        max_i, max_j = i - 1, j
    if j - 1 >= 0 and a[i][j - 1] > a[max_i][max_j]:
        max_i, max_j = i, j - 1
    if max_i != i or max_j != j:
        a[i][j], a[max_i][max_j] = a[max_i][max_j], a[i][j]
        maintain(max_i, max_j)

maintain(m - 1, n - 1)

```

- e. Using no other sorting method as a subroutine, show how to use an $n \times n$ Young tableau to sort n^2 numbers in $O(n^3)$ time.

```

def sort_elements(a):
    m = len(a)
    n = int(math.ceil(math.sqrt(m)))
    y = [[1e8 for _ in range(n)] for _ in range(n)]
    for val in a:
        insert(y, val)
    a = []
    for _ in range(m):
        a.append(extract_min(y))
    return a

```

INSERT and EXTRACT-MIN are $O(n)$, there are n^2 elements, therefore the result is $O(n^3)$.

- f. Give an $O(m + n)$ -time algorithm to determine whether a given number is stored in a given $m \times n$ Young tableau.

```
def find(a, val):
    m, n = len(a), len(a[0])
    i, j = 0, n - 1
    while i < m and j >= 0:
        if a[i][j] == val:
            return i, j
        elif a[i][j] > val:
            j -= 1
        else:
            i += 1
    return -1, -1
```

7 Quicksort

- 7.1 Description of quicksort
- 7.2 Performance of quicksort
- 7.3 A randomized version of quicksort
- 7.4 Analysis of quicksort
- Problems

7.1 Description of quicksort

7.1-1

Using Figure 7.1 as a model, illustrate the operation of PARTITION on the array
 $A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11 \rangle$

<https://github.com/CyberZH/G/>



<https://github.com/CyberZH/G/>



<https://github.com/CyberZH/G/>

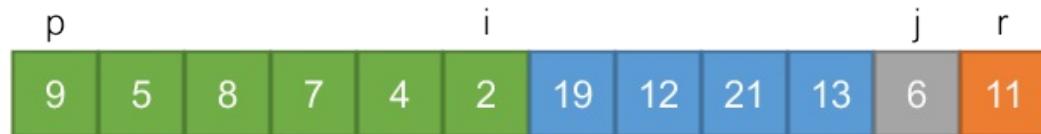
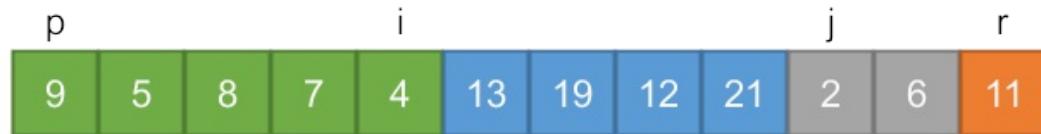
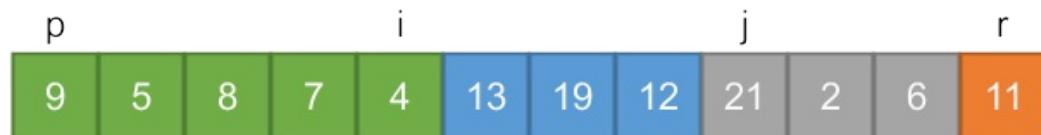
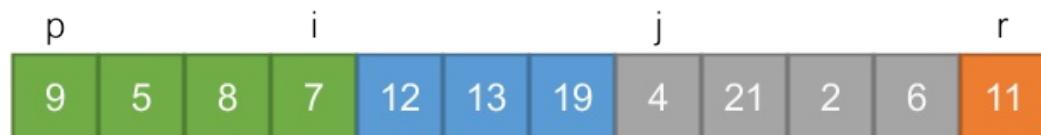


<https://github.com/CyberZH/G/>

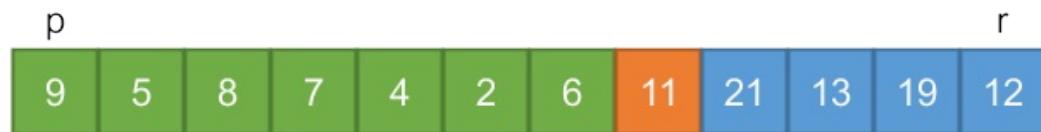


<https://github.com/CyberZH/G/>





<https://github.com/CyberZH/G/>



7.1-2

What value of q does PARTITION return when all elements in the array $A[p \dots r]$ have the same value? Modify PARTITION so that $q = \lfloor (p + r)/2 \rfloor$ when all elements in the array $A[p \dots r]$ have the same value.

PARTITION returns r .

```
def partition(a, p, r):
    x = a[r - 1]
    i = p - 1
    for k in range(p, r - 1):
        if a[k] < x:
            i += 1
            a[i], a[k] = a[k], a[i]
    i += 1
    a[i], a[r - 1] = a[r - 1], a[i]
    j = i
    for k in range(i + 1, r):
        if a[k] == x:
            j += 1
            a[j], a[k] = a[k], a[j]
            k -= 1
    return (i + j) // 2
```

7.1-3

Give a brief argument that the running time of PARTITION on a subarray of size n is $\Theta(n)$.

Only one loop.

7.1-4

How would you modify QUICKSORT to sort into nonincreasing order?

```
def partition(a, p, r):
    x = a[r - 1]
    i = p - 1
    for j in range(p, r - 1):
        if a[j] >= x:
            i += 1
            a[i], a[j] = a[j], a[i]
    i += 1
    a[i], a[r - 1] = a[r - 1], a[i]
    return i

def quicksort(a, p, r):
    if p < r - 1:
        q = partition(a, p, r)
        quicksort(a, p, q)
        quicksort(a, q + 1, r)
```

7.2 Performance of quicksort

7.2-1

Use the substitution method to prove that the recurrence

$T(n) = T(n - 1) + \Theta(n)$ has the solution $T(n) = \Theta(n^2)$, as claimed at the beginning of Section 7.2.

Suppose $T(n) \leq cn^2 - an$,

$$\begin{aligned} T(n) &\leq c(n-1)^2 - a(n-1) + dn \\ &= cn^2 + (d-2c)n + c \quad (d < 2c, n \geq \frac{c}{2c-d}) \\ &\leq cn^2 \end{aligned}$$

7.2-2

What is the running time of QUICKSORT when all elements of array A have the same value?

$$T(n) = T(n - 1) + \Theta(n)$$

$$T(n) = \Theta(n^2)$$

7.2-3

Show that the running time of QUICKSORT is $\Theta(n^2)$ when the array A contains distinct elements and is sorted in decreasing order.

$$T(n) = T(n - 1) + \Theta(n)$$

$$T(n) = \Theta(n^2)$$

7.2-4

Banks often record transactions on an account in order of the times of the transactions, but many people like to receive their bank statements with checks listed in order by check number. People usually write checks in order by check number, and merchants usually cash them with reasonable dispatch. The problem of converting time-of-transaction ordering to check-number ordering is therefore the problem of sorting almost-sorted input. Argue that the procedure INSERTION-SORT would tend to beat the procedure QUICKSORT on this problem.

INSERTION-SORT is $\Omega(n)$ while QUICKSORT is $\Omega(n^2)$

7.2-5

Suppose that the splits at every level of quicksort are in the proportion $1 - \alpha$ to α , where $0 < \alpha \leq 1/2$ is a constant. Show that the minimum depth of a leaf in the recursion tree is approximately $-\lg n / \lg \alpha$ and the maximum depth is approximately $-\lg n / \lg(1 - \alpha)$. (Don't worry about integer round-off.)

Let x be the minimum depth,

$$\begin{aligned} n\alpha^x &\leq 1 \\ \alpha^x &\leq n^{-1} \\ x &\geq \log_\alpha n^{-1} \\ x &\geq -\lg n / \lg \alpha \end{aligned}$$

Let y be the maximum depth,

$$\begin{aligned} n(1 - \alpha)^y &\leq 1 \\ (1 - \alpha)^y &\leq n^{-1} \\ y &\geq \log_{(1-\alpha)} n^{-1} \\ y &\geq -\lg n / \lg(1 - \alpha) \end{aligned}$$

7.2-6 *

Argue that for any constant $0 < \alpha \leq 1/2$, the probability is approximately $1 - 2\alpha$, that on a random input array, PARTITION produces a split more balanced than $1 - \alpha$ to α .

In order to make a partition which is less balanced, the pivot should belong to either the largest αn elements or the smallest αn elements. Thus a better partition is approximately

$$\frac{n - 2\alpha n}{n} = 1 - 2\alpha$$

7.3 A randomized version of quicksort

7.3-1

Why do we analyze the expected running time of a randomized algorithm and not its worst-case running time?

Even with the same input, the running time will be different.

7.3-2

When RANDOMIZED-QUICKSORT runs, how many calls are made to the random-number generator RANDOM in the worst case? How about in the best case? Give your answer in terms of Θ -notation.

Worst: $T(n) = T(n - 1) + \Theta(1) = \Theta(n)$

Best: $T(n) = 2T(n/2) + \Theta(1) = \Theta(n)$

7.4 Analysis of quicksort

7.4-1

Show that in the recurrence

$$T(n) = \max_{0 \leq q \leq n-1} (T(q) + T(n - q - 1)) + \Theta(n),$$

$$T(n) = \Omega(n^2).$$

Suppose $T(n) \geq cn^2$,

$$\begin{aligned} T(n) &\geq \max_{0 \leq q \leq n-1} (cq^2 + c(n - q - 1)^2) + dn \\ &\geq c(n - 1)^2 + dn && \left(q = \frac{n-1}{2} \right) \\ &= cn^2 + (d - 2c)n + c && (d > 2c) \\ &\geq cn^2 \\ &= \Omega(n^2) \end{aligned}$$

7.4-2

Show that quicksort's best-case running time is $\Omega(n \lg n)$.

$T(n) = 2T(n/2) + \Theta(n)$, therefore it is $\Omega(n \lg n)$.

7.4-3

Show that the expression $q^2 + (n - q - 1)^2$ achieves a maximum over $q = 0, 1, \dots, n - 1$ when $q = 0$ or $q = n - 1$.

Based on the first order derivation on q , we know the minimum is achieved when

$q = \frac{n-1}{2}$, and the function increases with the same speed when q is away from $\frac{n-1}{2}$

in two directions. Thus the maximum is on the bound of the the variable, $q = 0$ and $q = n - 1$.

7.4-4

Show that RANDOMIZED-QUICKSORT's expected running time is $\Omega(n \lg n)$.

$$\begin{aligned}
 E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-i+1} \\
 &= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
 &> \sum_{i=1}^{n-1} \sum_{k=1}^n \frac{1}{k} \\
 &= \sum_{i=1}^{n-1} \Omega(\lg n) \\
 &= n\Omega \lg n
 \end{aligned}$$

7.4-5

We can improve the running time of quicksort in practice by taking advantage of the fast running time of insertion sort when its input is "nearly" sorted. Upon calling quicksort on a subarray with fewer than k elements, let it simply return without sorting the subarray. After the top-level call to quicksort returns, run insertion sort on the entire array to finish the sorting process. Argue that this sorting algorithm runs in $O(nk + n \lg(n/k))$ expected time. How should we pick k , both in theory and in practice?

QUICK-SORT: layer number is $O(\lg(n/k))$, therefore it is $O(n \lg(n/k))$.

INSERTION-SORT: each subarray is $O(k^2)$, the number of subarrays is $O(n/k)$, therefore it is $O(nk)$.

Therefore this sorting algorithm runs in $O(nk + n \lg(n/k))$.

In practice we should use profiling.

7.4-6 *

Consider modifying the PARTITION procedure by randomly picking three elements from array A and partitioning about their median (the middle value of the three elements).

Approximate the probability of getting at worst an α -to- $(1 - \alpha)$ split, as a function of α in the range $0 < \alpha < 1$.

The worst case happens when at least two of the chosen elements are in the αn smallest or largest set, thus the probability of a worse case is

$$2 \left(\alpha^3 + \binom{3}{1} \alpha^2 (1 - \alpha) \right) = 6\alpha^2 - 4\alpha^3$$

The complementary is $1 - 6\alpha^2 + 4\alpha^3$.

Problems

7-1 Hoare partition correctness

The version of PARTITION given in this chapter is not the original partitioning algorithm. Here is the original partition algorithm, which is due to C. A. R. Hoare:

```

HOARE-PARTITION(A, p, r)
1  x = A[p]
2  i = p - 1
3  j = r + 1
4  while TRUE
5      repeat
6          j = j - 1
7          until A[j] <= x
8      repeat
9          i = i + 1
10         until A[i] >= x
11     if i < j
12         exchange A[i] with A[j]
13     else return j
    
```

a. Demonstrate the operation of HOARE-PARTITION on the array

$A = \langle 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21 \rangle$, showing the values of the array and auxiliary values after each iteration of the while loop in lines 4-13.

Initial state of array A:



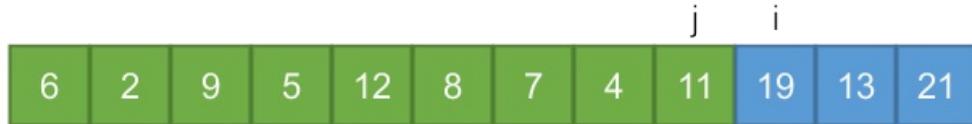
https://github.com/CyberZHG/



https://github.com/CyberZHG/



<https://github.com/Cyber2-HG>



The next three questions ask you to give a careful argument that the procedure HOARE-PARTITION is correct. Assuming that the subarray $A[p \dots r]$ contains at least two elements, prove the following:

- b. The indices i and j are such that we never access an element of A outside the subarray $A[p \dots r]$.

In the first loop, i will terminate at the pivot, the smallest j would be the pivot, therefore no invalid position is accessed. In the next loops, i will finally terminate at last j and i will finally terminate at last i , and since $i \geq p$ and $j \leq r$ after the first loop, there is no change to access an element outside $A[p \dots r]$.

- c. When HOARE-PARTITION terminates, it returns a value j such that $p \leq j < r$.

In b, we know $p \leq j \leq r$, the largest j in the first loop is r , while i will be at p , if $p \neq r$, then $i < r$, the loop will not terminate. In the second loop, j has to move at least one step, therefore j must be less than r .

- d. Every element of $A[p \dots j]$ is less than or equal to every element of $A[j + 1 \dots r]$ when HOARE-PARTITION terminates.

Small values are moved to the front and large values are moved to the end.

The PARTITION procedure in Section 7.1 separates the pivot value (originally in $A[r]$) from the two partitions it forms. The HOARE-PARTITION procedure, on the other hand, always places the pivot value (originally in $A[p]$) into one of the two partitions $A[p \dots j]$ and $A[j + 1 \dots r]$. Since $p \leq j < r$, this split is always nontrivial.

- e. Rewrite the QUICKSORT procedure to use HOARE-PARTITION.

```

def hoare_partition(a, p, r):
    x = a[p]
    i = p - 1
    j = r
    while True:
        while True:
            j -= 1
            if a[j] <= x:
                break
        while True:
            i += 1
            if a[i] >= x:
                break
        if i < j:
            a[i], a[j] = a[j], a[i]
        else:
            return j

def quicksort(a, p, r):
    if p < r - 1:
        q = hoare_partition(a, p, r)
        quicksort(a, p, q + 1)
        quicksort(a, q + 1, r)

```

7-2 Quicksort with equal element values

The analysis of the expected running time of randomized quicksort in Section 7.4.2 assumes that all element values are distinct. In this problem, we examine what happens when they are not.

- a. Suppose that all element values are equal. What would be randomized quicksort's running time in this case?

$$\Theta(n^2)$$

b. The PARTITION procedure returns an index q such that each element of $A[p \dots q - 1]$ is less than or equal to $A[q]$ and each element of $A[q + 1 \dots r]$ is greater than $A[q]$. Modify the PARTITION procedure to produce a procedure PARTITION'(A,p,r), which permutes the elements of $A[p \dots r]$ and returns two indices q and t , where $p \leq q \leq t \leq r$, such that

- all elements of $A[q \dots t]$ are equal,
- each element of $A[p \dots q - 1]$ is less than $A[q]$, and
- each element of $A[t + 1 \dots r]$ is greater than $A[q]$.

Like PARTITION, your PARTITION' procedure should take $\Theta(r - p)$ time.

```
def partition(a, p, r):
    x = a[r - 1]
    i = p - 1
    for k in range(p, r - 1):
        if a[k] < x:
            i += 1
            a[i], a[k] = a[k], a[i]
    i += 1
    a[i], a[r - 1] = a[r - 1], a[i]
    j = i
    for k in range(i + 1, r):
        if a[k] == x:
            j += 1
            a[j], a[k] = a[k], a[j]
    k -= 1
    return i, j
```

c. Modify the RANDOMIZED-QUICKSORT procedure to call PARTITION0, and name the new procedure RANDOMIZED-QUICKSORT'. Then modify the QUICKSORT procedure to produce a procedure QUICKSORT'(p, r) that calls RANDOMIZED-PARTITION' and recurses only on partitions of elements not known to be equal to each other.

```

def randomized_partition(a, p, r):
    x = random.randint(p, r - 1)
    a[x], a[r - 1] = a[r - 1], a[x]
    return partition(a, p, r)

def quicksort(a, p, r):
    if p < r - 1:
        q, t = randomized_partition(a, p, r)
        quicksort(a, p, q)
        quicksort(a, t + 1, r)

```

- d.** Using QUICKSORT', how would you adjust the analysis in Section 7.4.2 to avoid the assumption that all elements are distinct?

7-3 Alternative quicksort analysis

An alternative analysis of the running time of randomized quicksort focuses on the expected running time of each individual recursive call to RANDOMIZED-QUICKSORT, rather than on the number of comparisons performed.

- a.** Argue that, given an array of size n , the probability that any particular element is chosen as the pivot is $1/n$. Use this to define indicator random variables $X_i = I\{i^{\text{th}} \text{ smallest element is chosen as the pivot}\}$. What is $E[X_i]$?

$$E[X_i] = 1/n$$

- b.** Let $T(n)$ be a random variable denoting the running time of quicksort on an array of size n . Argue that

$$E[T(n)] = E \left[\sum_{q=1}^n X_q (T(q-1) + T(n-q) + \Theta(n)) \right]$$

Obviously.

- c.** Show that we can rewrite equation (7.5) as

$$E[T(n)] = \frac{2}{n} \sum_{q=2}^{n-1} E[T(q)] + \Theta(n)$$

$$\begin{aligned}
\mathbf{E}[T(n)] &= \mathbf{E} \left[\sum_{q=1}^n X_q (T(q-1) + T(n-q) + \Theta(n)) \right] \\
&= \frac{1}{n} \left[\sum_{q=1}^n \mathbf{E}[T(q-1)] + \sum_{q=1}^n \mathbf{E}[T(n-q)] \right] + \Theta(n) \\
&= \frac{1}{n} \left[\sum_{q=0}^{n-1} \mathbf{E}[T(q)] + \sum_{q=0}^{n-1} \mathbf{E}[T(q)] \right] + \Theta(n) \\
&= \frac{2}{n} \sum_{q=0}^{n-1} \mathbf{E}[T(q)] + \Theta(n) \\
&= \frac{2}{n} \sum_{q=2}^{n-1} \mathbf{E}[T(q)] + \Theta(n)
\end{aligned}$$

d. Show that

$$\begin{aligned}
\sum_{k=2}^{n-1} k \lg k &\leq \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 \\
\sum_{k=2}^{n-1} k \lg k &= \sum_{k=2}^{\lceil n/2 \rceil - 1} k \lg k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg k \\
&\leq \sum_{k=2}^{\lceil n/2 \rceil - 1} k \lg(n/2) + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg n \\
&= \frac{(1+n/2) \cdot n/2}{2} (\lg n - 1) + \frac{(n/2+n) \cdot n/2}{2} \lg n \\
&= \frac{n^2 + 2n}{8} (\lg n - 1) + \frac{3n^2}{8} \lg n \\
&= \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2
\end{aligned}$$

e. Using the bound from equation (7.7), show that the recurrence in equation (7.6) has the solution $\mathbf{E}[T(n)] = \Theta(n \lg n)$.

Suppose $\mathbf{E}[T(n)] \leq cn \lg n$,

$$\begin{aligned}
E[T(n)] &= \frac{2}{n} \sum_{q=2}^{n-1} E[T(q)] + \Theta(n) \\
&\leq \frac{2}{n} \sum_{q=2}^{n-1} cq \lg q + \Theta(n) \\
&\leq \frac{2c}{n} \left(\frac{1}{2} n^2 \lg n - \frac{1}{8} n^2 \right) + \Theta(n) \\
&= cn \lg n - \frac{1}{4} cn + \Theta(n) \\
&= \Theta(n \lg n)
\end{aligned}$$

7-4 Stack depth for quicksort

The QUICKSORT algorithm of Section 7.1 contains two recursive calls to itself. After QUICKSORT calls PARTITION, it recursively sorts the left subarray and then it recursively sorts the right subarray. The second recursive call in QUICKSORT is not really necessary; we can avoid it by using an iterative control structure. This technique, called tail recursion, is provided automatically by good compilers. Consider the following version of quicksort, which simulates tail recursion:

```

TAIL-RECURSIVE-QUICKSORT(A, p, r)
1 while p < r
2     // Partition and sort left subarray
3     q = PARTITION(A, p, r)
4     TAIL-RECURSIVE-QUICKSORT(A, p, q - 1)
5     p = q + 1

```

- a. Argue that $\text{TAIL-RECURSIVE-QUICKSORT}(A, 1, A.length)$ correctly sorts the array A .

The function needs to call $\text{QUICKSORT}(A, q + 1, r)$, set p to $q + 1$ then go to line 1 is exactly the same form of calling the function.

Compilers usually execute recursive procedures by using a stack that contains pertinent information, including the parameter values, for each recursive call. The information for the most recent call is at the top of the stack, and the information for the initial call is at the bottom. Upon calling a procedure, its information is pushed onto the stack; when it terminates, its information is popped. Since we assume that array parameters are represented by pointers, the information for each procedure call on the stack requires $O(1)$ stack space. The stack depth is the maximum amount of stack space used at any time during a computation.

- b.** Describe a scenario in which TAIL-RECURSIVE-QUICKSORT's stack depth is $\Theta(n)$ on an n -element input array.

$$T(n) = T(1) + T(n - 1) + \Theta(n)$$

- c.** Modify the code for TAIL-RECURSIVE-QUICKSORT so that the worst-case stack depth is $\Theta(\lg n)$. Maintain the $O(n \lg n)$ expected running time of the algorithm.

```
def partition(a, p, r):
    x = a[r - 1]
    i = p - 1
    for k in range(p, r - 1):
        if a[k] < x:
            i += 1
            a[i], a[k] = a[k], a[i]
    i += 1
    a[i], a[r - 1] = a[r - 1], a[i]
    j = i
    for k in range(i + 1, r):
        if a[k] == x:
            j += 1
            a[j], a[k] = a[k], a[j]
            k -= 1
    return i, j

def randomized_partition(a, p, r):
    x = random.randint(p, r - 1)
    a[x], a[r - 1] = a[r - 1], a[x]
    return partition(a, p, r)

def quicksort(a, p, r):
    while p < r - 1:
        q, t = randomized_partition(a, p, r)
        if q - p < r - t:
            quicksort(a, p, q)
            p = t + 1
        else:
            quicksort(a, t + 1, r)
            r = q
```

7-5 Median-of-3 partition

One way to improve the RANDOMIZED-QUICKSORT procedure is to partition around a pivot that is chosen more carefully than by picking a random element from the subarray. One common approach is the median-of-3 method: choose the pivot as the median (middle element) of a set of 3 elements randomly selected from the subarray. (See Exercise 7.4-6.) For this problem, let us assume that the elements in the input array $A[1 \dots n]$ are distinct and that $n \geq 3$. We denote the sorted output array by $A'[1 \dots n]$. Using the median-of-3 method to choose the pivot element x , define $p_i = \Pr\{x = A'[i]\}$.

- a.** Give an extract formula for p_i as a function of n and i for $i = 2, 3, \dots, n - 1$. (Note that $p_1 = p_n = 0$.)

$$p_i = \binom{3}{1} \frac{1}{n} \cdot \binom{2}{1} \frac{i-1}{n-1} \cdot \frac{n-i}{n-2} = \frac{6(i-1)(n-i)}{n(n-1)(n-2)}$$

- b.** By what amount have we increased the likelihood of choosing the pivot as $x = A'[\lfloor (n+1)/2 \rfloor]$, the median of $A[1 \dots n]$, compared with the ordinary implementation? Assume that $n \rightarrow \infty$, and give the limiting ratio of these probabilities.

$$p_{\lfloor (n+1)/2 \rfloor} \approx \frac{6(\frac{n+1}{2} - 1)(n - \frac{n+1}{2})}{n(n-1)(n-2)} = \frac{3(n-1)}{2n(n-2)}$$

$$\lim_{n \rightarrow \infty} \frac{\frac{3(n-1)}{2n(n-2)}}{\frac{1}{n}} = \lim_{n \rightarrow \infty} \frac{3(n-1)}{2(n-2)} = \frac{3}{2}$$

- c.** If we define a "good" split to mean choosing the pivot as $x = A'[i]$, where $n/3 \leq i \leq 2n/3$, by what amount have we increased the likelihood of getting a good split compared with the ordinary implementation?

$$\begin{aligned}
\lim_{n \rightarrow \infty} \sum_{i=n/3}^{2n/3} \frac{6(i-1)(n-i)}{n(n-1)(n-2)} &= \lim_{n \rightarrow \infty} \frac{6}{n(n-1)(n-2)} \sum_{i=n/3}^{2n/3} (i-1)(n-i) \\
&\approx \lim_{n \rightarrow \infty} \frac{6}{n(n-1)(n-2)} \int_{n/3}^{2n/3} (x-1)(n-x) dx \\
&= \lim_{n \rightarrow \infty} \frac{6}{n(n-1)(n-2)} \left(-\frac{1}{3}x^3 + \frac{1}{2}(n+1)x^2 - nx \right) \Big|_{n/3}^{2n/3} \\
&= \lim_{n \rightarrow \infty} \frac{6}{n(n-1)(n-2)} \left(-\frac{7}{81}n^3 + \frac{1}{6}n^3 - \frac{1}{6}n^2 \right) \\
&= \lim_{n \rightarrow \infty} \frac{1}{n(n-1)(n-2)} \left(-\frac{14}{27}n^3 + n^3 - n^2 \right) \\
&= \lim_{n \rightarrow \infty} \frac{\frac{13}{27}n^3 - n^2}{n(n-1)(n-2)} \\
&= \frac{13}{27}
\end{aligned}$$

d. Argue that in the $\Omega(n \lg n)$ running time of quicksort, the median-of-3 method affects only the constant factor.

Even if median-of-3 choose the median of $A[p \dots r]$, the running time is still $T(n) = 2T(n/2) + \Theta(n)$, which is $\Omega(n \lg n)$.

7-6 Fuzzy sorting of intervals

Consider a sorting problem in which we do not know the numbers exactly. Instead, for each number, we know an interval on the real line to which it belongs. That is, we are given n closed intervals of the form $[a_i, b_i]$, where $a_i \leq b_i$. We wish to **fuzzy-sort** these intervals, i.e., to produce a permutation $\langle i_1, i_2, \dots, i_n \rangle$ of the intervals such that for $j = 1, 2, \dots, n$, there exist $c_j \in [a_{i_j}, b_{i_j}]$ satisfying $c_1 \leq c_2 \leq \dots \leq c_n$.

a. Design a randomized algorithm for fuzzy-sorting n intervals. Your algorithm should have the general structure of an algorithm that quicksorts the left endpoints (the a_i values), but it should take advantage of overlapping intervals to improve the running time. (As the intervals overlap more and more, the problem of fuzzy-sorting the intervals becomes progressively easier. Your algorithm should take advantage of such overlapping, to the extent that it exists.)

Find the intervals that all have a common overlapping with the pivot, these intervals could be seen as equal since there is a c in the common overlapping. The following is the same as 7.2.

```

class Interval:
    def __init__(self, l, r):
        self.l = l
        self.r = r

    def __lt__(self, other):
        return self.r < other.l

    def __str__(self):
        return '(' + str(self.l) + ', ' + str(self.r) + ')'

    def get_intersect(self, interval):
        return Interval(max(self.l, interval.l), min(self.r, interval.r))

def partition(a, p, r):
    x = a[r - 1]
    for k in range(p, r - 1):
        next_x = x.get_intersect(a[k])
        if next_x.l <= next_x.r:
            x = next_x
    i = p - 1
    for k in range(p, r - 1):
        if a[k] < x:
            i += 1
            a[i], a[k] = a[k], a[i]
    i += 1
    a[i], a[r - 1] = a[r - 1], a[i]
    j = i
    inter = a[i]
    for k in range(i + 1, r):
        next_x = x.get_intersect(a[k])
        if next_x.l <= next_x.r:
            j += 1
            a[j], a[k] = a[k], a[j]
            k -= 1
    return i, j

def randomized_partition(a, p, r):
    x = random.randint(p, r - 1)
    a[x], a[r - 1] = a[r - 1], a[x]
    return partition(a, p, r)

def quicksort(a, p, r):
    if p < r - 1:
        q, t = randomized_partition(a, p, r)
        quicksort(a, p, q)
        quicksort(a, t + 1, r)

```

- b.** Argue that your algorithm runs in expected time $\Theta(n \lg n)$ in general, but runs in expected time $\Theta(n)$ when all of the intervals overlap (i.e., when there exists a value x such that $x \in [a_i, b_i]$ for all i). Your algorithm should not be checking for this case explicitly; rather, its performance should naturally improve as the amount of overlap increases.

The algorithm is based on quick-sort, therefore it is $\Theta(n \lg n)$.

If all of the intervals overlap, the partition returns $(1, n)$ immediately, there is no need for further recursion. Thus the expected time is the expected time of partition, which is $\Theta(n)$.

8 Sorting in Linear Time

- 8.1 Lower bounds for sorting
- 8.2 Counting sort
- 8.3 Radix sort
- 8.4 Bucket sort
- Problems

8.1 Lower bounds for sorting

8.1-1

What is the smallest possible depth of a leaf in a decision tree for a comparison sort?

For a permutation $a_1 \leq a_2 \leq \dots \leq a_n$, there are $n - 1$ pairs of relative ordering, thus the smallest possible depth is $n - 1$.

8.1-2

Obtain asymptotically tight bounds on $\lg(n!)$ without using Stirling's approximation.

Instead, evaluate the summation $\sum_{k=1}^n \lg k$ using techniques from Section A.2.

$$\begin{aligned}\sum_{k=1}^n \lg k &\leq \sum_{k=1}^n \lg n \\ &= n \lg n\end{aligned}$$

$$\begin{aligned}\sum_{k=1}^n \lg k &= \sum_{k=2}^{n/2} \lg k + \sum_{k=n/2}^n \lg k \\ &\geq \sum_{k=1}^{n/2} 1 + \sum_{k=n/2}^n \lg n/2 \\ &= \frac{n}{2} + \frac{n}{2} (\lg n - 1) \\ &= \frac{n}{2} \lg n\end{aligned}$$

8.1-3

Show that there is no comparison sort whose running time is linear for at least half of the $n!$ inputs of length n . What about a fraction of $1/n$ of the inputs of length n ?

What about a fraction $1/2^n$?

$$\begin{aligned}
 \frac{n!}{2} &\leq 2^h \\
 h &\geq \lg\left(\frac{n!}{2}\right) \\
 &= (\lg n!) - 1 \\
 &= \Omega(n \lg n) - 1 \\
 &= \Omega(n \lg n)
 \end{aligned}$$

$$\begin{aligned}
 \frac{n!}{n} &\leq 2^h \\
 h &\geq \lg\left(\frac{n!}{n}\right) \\
 &= (\lg n!) - \lg n \\
 &= \Omega(n \lg n) - \lg n \\
 &= \Omega(n \lg n)
 \end{aligned}$$

$$\begin{aligned}
 \frac{n!}{2^n} &\leq 2^h \\
 h &\geq \lg\left(\frac{n!}{2^n}\right) \\
 &= (\lg n!) - n \\
 &= \Omega(n \lg n) - n \\
 &= \Omega(n \lg n)
 \end{aligned}$$

All of them have the lower-bound $\Omega(n \lg n)$.

8.1-4

Suppose that you are given a sequence of n elements to sort. The input sequence consists of n/k subsequences, each containing k elements. The elements in a given subsequence are all smaller than the elements in the succeeding subsequence and larger than the elements in the preceding subsequence. Thus, all that is needed to sort the whole sequence of length n is to sort the k elements in each of the n/k subsequences. Show an $\Omega(n \lg k)$ lower bound on the number of comparisons needed to solve this variant of the sorting problem.

$$\begin{aligned}(k!)^{n/k} &\leq 2^h \\ n/k \lg(k!) &\leq h \\ kh/n &\geq \lg(k!) \\ kh/n &= \Omega(k \lg k) \\ h &= \Omega(n \lg k)\end{aligned}$$

8.2 Counting sort

8.2-1

Using Figure 8.2 as a model, illustrate the operation of COUNTING-SORT on the array

$$A = \langle 6, 0, 2, 0, 1, 3, 4, 6, 1, 3, 2 \rangle$$

8.2-2

Prove that COUNTING-SORT is stable.

Value with larger index choose the largest index first.

8.2-3

Suppose that we were to rewrite the for loop header in line 10 of the COUNTING-SORT as

```
10 for j = 1 to A.length
```

Show that the algorithm still works properly. Is the modified algorithm stable?

works properly but not stable.

8.2-4

Describe an algorithm that, given n integers in the range 0 to k , preprocesses its input and then answers any query about how many of the n integers fall into a range $[a \dots b]$ in $O(1)$ time. Your algorithm should use $\Theta(n + k)$ preprocessing time.

Use c in the counting sort, the number of integers fall into a range $[a \dots b]$ is $C[b] - C[a - 1]$.

```
class CountInterval:
    def __init__(self, a):
        k = max(a)
        self.c = [0 for _ in range(k + 1)]
        for v in a:
            self.c[v] += 1
        for i in range(1, k + 1):
            self.c[i] += self.c[i - 1]

    def count(self, a, b):
        if a == 0:
            return self.c[b]
        return self.c[b] - self.c[a - 1]
```

8.3 Radix sort

8.3-1

Using Figure 8.3 as a model, illustrate the operation of RADIX-SORT on the following list of English words: COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR, TAR, DIG, BIG, TEA, NOW, FOX.

0	1	2	3
COW	SEA	TAB	BAR
DOG	TEA	BAR	BIG
SEA	MOB	EAR	BOX
RUG	TAB	TAR	COW
ROW	DOG	SEA	DIG
MOB	RUG	TEA	DOG
BOX	DIG	DIG	EAR
TAB	BIG	BIG	FOX
BAR	BAR	MOB	MOB
EAR	EAR	DOG	NOW
TAR	TAR	COW	ROW
DIG	COW	ROW	RUG
BIG	ROW	NOW	SEA
TEA	NOW	BOX	TAB
NOW	BOX	FOX	TAR
FOX	FOX	RUG	TEA

8.3-2

Which of the following sorting algorithms are stable: insertion sort, merge sort, heapsort, and quicksort? Give a simple scheme that makes any sorting algorithm stable. How much additional time and space does your scheme entail?

Stable: insertion sort, merge sort.

When two values are equals, compare the original index. Additional space: $\Theta(n)$

8.3-3

Use induction to prove that radix sort works. Where does your proof need the assumption that the intermediate sort is stable?

...

8.3-4

Show how to sort n integers in the range 0 to $n^3 - 1$ in $O(n)$ time.

n -ary radix sort, including three $O(n)$ counting sort.

8.3-5 *

In the first card-sorting algorithm in this section, exactly how many sorting passes are needed to sort d -digit decimal numbers in the worst case? How many piles of cards would an operator need to keep track of in the worst case?

$\Theta(k^d)$ passes.

$\Theta(nk)$ piles.

8.4 Bucket sort

8.4-1

Using Figure 8.4 as a model, illustrate the operation of BUCKET-SORT on the array $A = \langle .79, .13, .16, .64, .39, .20, .89, .53, .71, .42 \rangle$.

R	
0	
1	.13 .16
2	.20
3	.39
4	.42
5	.53
6	.64
7	
8	.79 .71
9	.89

$$A = \langle .13, .16, .20, .39, .42, .53, .64, .71, .79, .89 \rangle$$

8.4-2

Explain why the worst-case running time for bucket sort is $\Theta(n^2)$. What simple change to the algorithm preserves its linear average-case running time and makes its worst-case running time $O(n \lg n)$?

Worst: all the elements falls in one bucket, $\Theta(n^2)$ sorting.

Change: use merge sort in each bucket.

8.4-3

Let X be a random variable that is equal to the number of heads in two flips of a fair coin. What is $E[X^2]$? What is $E^2[X]$?

$$\mathbb{E}[X] = 2 \cdot \frac{1}{4} + 1 \cdot \frac{1}{2} + 0 \cdot \frac{1}{4} = 1$$

$$\mathbb{E}[X^2] = 4 \cdot \frac{1}{4} + 1 \cdot \frac{1}{2} + 0 \cdot \frac{1}{4} = 1.5$$

$$\mathbb{E}^2[X] = \mathbb{E}[X] \cdot \mathbb{E}[X] = 1 \cdot 1 = 1$$

8.4-4 *

We are given n points in the unit circle, $p_i = (xi, yi)$, such that $0 < x_i^2 + y_i^2 \leq 1$ for $i = 1, 2, \dots, n$. Suppose that the points are uniformly distributed; that is, the probability of finding a point in any region of the circle is proportional to the area of that region. Design an algorithm with an average-case running time of $\Theta(n)$ to sort the n

points by their distances $d_i = \sqrt{x_i^2 + y_i^2}$ from the origin.

Bucket sort by radius,

$$\pi r_i^2 = \frac{i}{10} \cdot \pi 1^2$$

$$r_i = \sqrt{\frac{i}{10}}$$

8.4-5 *

A **probability distribution function** $P(x)$ for a random variable X is defined by $P(x) = \Pr\{X \leq x\}$. Suppose that we draw a list of n random variables X_1, X_2, \dots, X_n from a continuous probability distribution function P that is computable in $O(1)$ time. Give an algorithm that sorts these numbers in linear average-case time.

Bucket sort by p_i ,

$$P(p_i) = \frac{i}{10}$$

Problems

8-1 Probabilistic lower bounds on comparison sorting

In this problem, we prove a probabilistic $\Omega(n \lg n)$ lower bound on the running time of any deterministic or randomized comparison sort on n distinct input elements. We begin by examining a deterministic comparison sort A with decision tree T_A . We assume that every permutation of A 's inputs is equally likely.

- a.** Suppose that each leaf of T_A is labeled with the probability that it is reached given a random input. Prove that exactly $n!$ leaves are labeled $1/n!$ and that the rest are labeled 0.

There should be only $n!$ leaves.

- b.** Let $D(T)$ denote the external path length of a decision tree T ; that is, $D(T)$ is the sum of the depths of all the leaves of T . Let T be a decision tree with $k > 1$ leaves, and let LT and RT be the left and right subtrees of T . Show that

$$D(T) = D(LT) + D(RT) + k$$

Add T means all the k depths of leaves increase by 1.

- c.** Let $d(k)$ be the minimum value of $D(T)$ over all decision trees T with $k > 1$ leaves. Show that $d(k) = \min_{1 \leq i \leq k-1} \{d(i) + d(k-i) + k\}$.

$D(T) = D(LT) + D(RT) + k$, $d(k) = \min_{1 \leq i \leq k-1} \{d(i) + d(k-i) + k\}$ iterates all the possibilities.

- d.** Prove that for a given value of $k > 1$ and i in the range $1 \leq i \leq k-1$, the function $i \lg i + (k-i) \lg (k-i)$ is minimized at $i = k/2$. Conclude that $d(k) = \Omega(k \lg k)$.

$$\begin{aligned}
 f(i) &= i \lg i + (k - i) \lg(k - i) \\
 f'(i) &= \lg i - \lg(k - i) \\
 0 &= \lg \frac{i}{k - i} \\
 1 &= \frac{i}{k - i} \\
 i &= k/2
 \end{aligned}$$

e. Prove that $D(T_A) = \Omega(n! \lg(n!))$, and conclude that the average-case time to sort n elements is $\Omega(n \lg n)$.

$$d(T_A) = d(n!) = \Omega(n! \lg n!)$$

Average:

$$\frac{\Omega(n! \lg n!)}{n!} = \Omega(\lg n!) = \Omega(n \lg n)$$

Now, consider a *randomized* comparison sort B . We can extend the decision-tree model to handle randomization by incorporating two kinds of nodes: ordinary comparison nodes and "randomization" nodes. A randomization node models a random choice of the form $\text{RANDOM}(1, r)$ made by algorithm B ; the node has r children, each of which is equally likely to be chosen during an execution of the algorithm.

f. Show that for any randomized comparison sort B , there exists a deterministic comparison sort A whose expected number of comparisons is no more than those made by B .

...

8-2 Sorting in place in linear time

Suppose that we have an array of n data records to sort and that the key of each record has the value 0 or 1. An algorithm for sorting such a set of records might possess some subset of the following three desirable characteristics:

1. The algorithm runs in $O(n)$ time.
2. The algorithm is stable.
3. The algorithm sorts in place, using no more than a constant amount of storage space in addition to the original array.

a. Give an algorithm that satisfies criteria 1 and 2 above.

Counting sort.

b. Give an algorithm that satisfies criteria 1 and 3 above.

Partition.

c. Give an algorithm that satisfies criteria 2 and 3 above.

Insertion sort.

d. Can you use any of your sorting algorithms from parts (a)-(c) as the sorting method used in line 2 of RADIX-SORT, so that RADIX-SORT sorts n records with b -bit keys in $O(bn)$ time? Explain how or why not.

First, stable and quick.

e. Suppose that the n records have keys in the range from 1 to k . Show how to modify counting sort so that it sorts the records in place in $O(n + k)$ time. You may use $O(k)$ storage outside the input array. Is your algorithm stable?

Same as permutation group:

```
def counting_in_place(a):
    k = max(a)
    c = [0 for _ in range(k + 1)]
    for v in a:
        c[v] += 1
    for i in range(1, k + 1):
        c[i] += c[i - 1]
    r = c[:]
    for i in range(len(a)):
        while True:
            if a[i] == 0:
                if i < r[0]:
                    break
            else:
                if r[a[i] - 1] <= i < r[a[i]]:
                    break
            c[a[i]] -= 1
            pos = c[a[i]]
            a[i], a[pos] = a[pos], a[i]
```

Not stable.

8-3 Sorting variable-length items

- a. You are given an array of integers, where different integers may have different numbers of digits, but the total number of digits over all the integers in the array is n . Show how to sort the array in $O(n)$ time.

Suppose the number of integers which have b_i digits is n_i , divide the integers into different buckets using counting sort, the integers in the same bucket have the same b_i , then use radix sort in each bucket:

$$\sum_i b_i n_i = n$$

therefore the algorithm is $O(n)$.

```

def counting_sort(a, m):
    b = [0 for _ in range(len(a))]
    k = 10
    c = [0 for _ in range(k)]
    for s in a:
        c[ord(s[m]) - ord('0')] += 1
    for i in range(1, k):
        c[i] += c[i - 1]
    for i in range(len(a) - 1, -1, -1):
        c[ord(a[i][m]) - ord('0')] -= 1
        b[c[ord(a[i][m]) - ord('0')]] = a[i]
    return b

def radix_sort(a):
    for m in range(len(a[0]) - 1, -1, -1):
        a = counting_sort(a, m)
    return a

def count_and_divide(a):
    a = map(str, a)
    b = [0 for _ in range(len(a))]
    k = 0
    for s in a:
        k = max(k, len(s))
    c = [0 for _ in range(k + 1)]
    for s in a:
        c[len(s)] += 1
    for i in range(1, k + 1):
        c[i] += c[i - 1]
    r = c[:]
    for i in range(len(a) - 1, -1, -1):
        c[len(a[i])] -= 1
        b[c[len(a[i])]] = a[i]
    for i in range(k + 1):
        if c[i] < r[i]:
            b[c[i]:r[i]] = radix_sort(b[c[i]:r[i]])
    return map(int, b)

```

- b.** You are given an array of strings, where different strings may have different numbers of characters, but the total number of characters over all the strings is n . Show how to sort the strings in $O(n)$ time. (Note that the desired order here is the standard alphabetical order; for example, $a < ab < b$.)

Sort the strings by their first characters with counting-sort, then divide the strings by their first characters, repeat the process in each new group. Since each character is used only once for sorting, the amortized running time is $O(n)$.

```

def get_key(s, i):
    if i >= len(s):
        return 0
    return ord(s[i]) - ord('a') + 1

def counting_sort(a, p=0):
    k = 27
    b = ['' for _ in range(len(a))]
    c = [0 for _ in range(k)]
    for s in a:
        c[get_key(s, p)] += 1
    for i in range(1, k):
        c[i] += c[i - 1]
    r = c[:]
    for i in range(len(a) - 1, -1, -1):
        c[get_key(a[i], p)] -= 1
        b[c[get_key(a[i], p)]] = a[i]
    for i in range(1, k):
        if c[i] < r[i]:
            b[c[i]:r[i]] = counting_sort(b[c[i]:r[i]], p+1)
    return b

```

8-4 Water jugs

Suppose that you are given n red and n blue water jugs, all of different shapes and sizes. All red jugs hold different amounts of water, as do the blue ones. Moreover, for every red jug, there is a blue jug that holds the same amount of water, and vice versa.

Your task is to find a grouping of the jugs into pairs of red and blue jugs that hold the same amount of water. To do so, you may perform the following operation: pick a pair of jugs in which one is red and one is blue, fill the red jug with water, and then pour the water into the blue jug. This operation will tell you whether the red or the blue jug can hold more water, or that they have the same volume. Assume that such a comparison takes one time unit. Your goal is to find an algorithm that makes a minimum number of comparisons to determine the grouping. Remember that you may not directly compare two red jugs or two blue jugs.

a. Describe a deterministic algorithm that uses $\Theta(n^2)$ comparisons to group the jugs into pairs.

Compare each red jug with each blue jug.

b. Prove a lower bound of $\Omega(n \lg n)$ for the number of comparisons that an algorithm solving this problem must make.

$$\begin{aligned} n! &\leq 3^h \\ h &= \Omega(n \lg n) \end{aligned}$$

- c. Give a randomized algorithm whose expected number of comparisons is $O(n \lg n)$, and prove that this bound is correct. What is the worst-case number of comparisons for your algorithm?

Random choose a red jug as pivot and partition blue rugs, and use the blue rug which is equal to the red rug as pivot and partition red rugs.

Worst case is $O(n^2)$.

```
def partition(a, b, p, r):
    pos = random.randint(p, r - 1)
    i = p - 1
    for j in range(p, r):
        if b[j] <= a[pos]:
            i += 1
            b[i], b[j] = b[j], b[i]
        if b[i] == a[pos]:
            k = i
    b[i], b[k] = b[k], b[i]
    pos = i
    i = p - 1
    for j in range(p, r):
        if a[j] <= b[pos]:
            i += 1
            a[i], a[j] = a[j], a[i]
        if a[i] == b[pos]:
            k = i
    a[i], a[k] = a[k], a[i]
    return pos

def quick_sort(a, b, p, r):
    if p + 1 < r:
        q = partition(a, b, p, r)
        quick_sort(a, b, p, q)
        quick_sort(a, b, q + 1, r)
```

8-5 Average sorting

Suppose that, instead of sorting an array, we just require that the elements increase on average. More precisely, we call an n -element array A **k -sorted** if, for all $i = 1, 2, \dots, n - k$, the following holds:

$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \leq \frac{\sum_{j=i+1}^{i+k} A[j]}{k}$$

a. What does it mean for an array to be 1-sorted?

Sorted.

b. Give a permutation of the numbers $1, 2, \dots, 10$ that is 2-sorted, but not sorted.

1, 2, 1, 2, 1, 2

c. Prove that an n -element array is k -sorted if and only if $A[i] \leq A[i + k]$ for all $i = 1, 2, \dots, n - k$.

$$\begin{aligned} \frac{\sum_{j=i}^{i+k-1} A[j]}{k} &\leq \frac{\sum_{j=i+1}^{i+k} A[j]}{k} \\ \sum_{j=i}^{i+k-1} A[j] &\leq \sum_{j=i+1}^{i+k} A[j] \\ A[i] &\leq A[i + k] \end{aligned}$$

d. Give an algorithm that k -sorts an n -element array in $O(n \lg(n/k))$ time.

We need to sort k groups, each group has the same $(i \bmod k)$. We can sort one group in $O(n/k \lg(n/k))$, to sort k groups, it is $O(n \lg(n/k))$.

```
def k_sort(a, k):
    for i in range(k):
        a[i:len(a):k] = sorted(a[i:len(a):k])
```

We can also show a lower bound on the time to produce a k -sorted array, when k is a constant.

e. Show that we can sort a k -sorted array of length n in $O(n \lg k)$ time.

Same as Exercise 6.5-9.

- f.** Show that when k is a constant, k -sorting an n -element array requires $\Omega(n \lg n)$ time.

$$\Omega(n \lg n/k) = \Omega(n \lg n)$$

8-6 Lower bound on merging sorted lists

The problem of merging two sorted lists arises frequently. We have seen a procedure for it as the subroutine MERGE in Section 2.3.1. In this problem, we will prove a lower bound of $2n - 1$ on the worst-case number of comparisons required to merge two sorted lists, each containing n items. First we will show a lower bound of $2n - o(n)$ comparisons by using a decision tree.

- a.** Given $2n$ numbers, compute the number of possible ways to divide them into two sorted lists, each with n numbers.

$$\binom{2n}{n}$$

- b.** Using a decision tree and your answer to part (a), show that any algorithm that correctly merges two sorted lists must perform at least $2n - o(n)$ comparisons.

Based on Exercise C.1.13,

$$\begin{aligned} \binom{2n}{n} &\leq 2^h \\ h &\geq \lg \frac{(2n)!}{(n!)^2} \\ &= \lg(2n!) - 2\lg(n!) \\ &= \Theta(2n \lg 2n) - 2\Theta(n \lg n) \\ &= \Theta(2n) \end{aligned}$$

Now we will show a slightly tighter $2n - 1$ bound.

- c.** Show that if two elements are consecutive in the sorted order and from different lists, then they must be compared.

We have to know the order of the two consecutive elements.

- d.** Use your answer to the previous part to show a lower bound of $2n - 1$ comparisons for merging two sorted lists.

There are $2n - 1$ pairs of consecutive elements.

8-7 The 0-1 sorting lemma and columnsort

...

9 Medians and Order Statistics

- 9.1 Minimum and maximum
- 9.2 Selection in expected linear time
- 9.3 Selection in worst-case linear time
- Problems

9.1 Minimum and maximum

9.1-1

Show that the second smallest of n elements can be found with $n + \lceil \lg n \rceil - 2$ comparisons in the worst case.

Divide the elements into the leaves of a binary tree. In each node, we compare the minimum values of its two sub-trees, then in the root node we know which is the smallest element using $n - 1$ comparisons. Since only the smallest element is less than the second smallest element, the two elements must have been compared in order to knock out the second smallest element when finding the minimum. In other words, the second smallest number must have been appeared as the opponent in the path to the leaf which has the smallest element. The depth of the tree is $\lceil \lg n \rceil$, thus we need $\lceil \lg n \rceil - 1$ comparisons to find the second smallest element.

```
def find_second_smallest(a, l, r):
    if l + 1 == r:
        return a[1], []
    mid = (l + r + 1) // 2
    min_l, lst_l = find_second_smallest(a, l, mid)
    min_r, lst_r = find_second_smallest(a, mid, r)
    if min_l <= min_r:
        min_val, lst = min_l, lst_l + [min_r]
    else:
        min_val, lst = min_r, lst_r + [min_l]
    if l == 0 and r == len(a):
        idx = 0
        for i in range(1, len(lst)):
            if lst[i] < lst[idx]:
                idx = i
        return lst[idx]
    return min_val, lst
```

9.1-2 *

Prove the lower bound of $\lceil 3n/2 \rceil - 2$ comparisons in the worst case to find both the maximum and minimum of n numbers.

If n is odd, there are

$$1 + \frac{3(n-3)}{2} + 2 = \frac{3n}{2} - \frac{3}{2} = \left(\left\lceil \frac{3n}{2} \right\rceil - \frac{1}{2} \right) - \frac{3}{2} = \left\lceil \frac{3n}{2} \right\rceil - 2$$

comparisons.

If n is even, there are

$$1 + \frac{3(n-2)}{2} = \frac{3n}{2} - 2 = \left\lceil \frac{3n}{2} \right\rceil - 2$$

comparisons.

9.2 Selection in expected linear time

9.2-1

Show that RANDOMIZED-SELECT never makes a recursive call to a 0-length array.

...

9.2-2

Argue that the indicator random variable X_k and the value $T(\max(k - 1, n - k))$ are independent.

...

9.2-3

Write an iterative version of RANDOMIZED-SELECT.

```

def partition(a, p, r):
    x = a[r - 1]
    i = p - 1
    for k in range(p, r - 1):
        if a[k] < x:
            i += 1
            a[i], a[k] = a[k], a[i]
    i += 1
    a[i], a[r - 1] = a[r - 1], a[i]
    return i

def randomized_partition(a, p, r):
    x = random.randint(p, r - 1)
    a[x], a[r - 1] = a[r - 1], a[x]
    return partition(a, p, r)

def randomized_select(a, p, r, i):
    while True:
        if p + 1 == r:
            return a[p]
        q = randomized_partition(a, p, r)
        k = q - p + 1
        if i == k:
            return a[q]
        if i < k:
            r = q
        else:
            p = q + 1
            i -= k

```

9.2-4

Suppose we use RANDOMIZED-SELECT to select the minimum element of the array $A = \langle 3; 2; 9; 0; 7; 5; 4; 8; 6; 1 \rangle$. Describe a sequence of partitions that results in a worst-case performance of RANDOMIZED-SELECT.

Select 9, 8, 7, 6, 5, 4, 3, 2, 1.

9.3 Selection in worst-case linear time

9.3-1

In the algorithm SELECT, the input elements are divided into groups of 5. Will the algorithm work in linear time if they are divided into groups of 7? Argue that SELECT does not run in linear time if groups of 3 are used.

Suppose the input elements are divided into 7 groups, then

$$4 \left(\left\lceil \frac{1}{2} \left\lceil \frac{n}{7} \right\rceil \right\rceil - 2 \right) \geq \frac{2n}{7} - 8$$

$$T(n) = T(\lceil n/7 \rceil) + T(5n/7 + 8) + O(n)$$

Suppose $T(n) \leq cn$,

$$\begin{aligned} T(n) &\leq cn/7 + c + 8c + 5cn/7 + an \\ &= 6cn/7 + 9c + an \\ &= cn + (-cn/7 + 9c + an) \\ &\leq cn \end{aligned}$$

Suppose the input elements are divided into 3 groups, then

$$2 \left(\left\lceil \frac{1}{2} \left\lceil \frac{n}{3} \right\rceil \right\rceil - 2 \right) \geq \frac{n}{3} - 4$$

$$T(n) = T(\lceil n/3 \rceil) + T(2n/3 + 4) + O(n)$$

Suppose $T(n) \geq cn$,

$$\begin{aligned} T(n) &\geq cn/3 + c + 4c + 2cn/3 + an \\ &= cn + 5c + an \\ &> cn \end{aligned}$$

Therefore SELECT does not run in linear time if groups of 3 are used.

9.3-2

Analyze SELECT to show that if $n \geq 140$, then at least $\lceil n/4 \rceil$ elements are greater than the median-of-medians x and at least $\lceil n/4 \rceil$ elements are less than x .

$$\begin{aligned}\frac{3n}{10} - 6 &\geq \left\lceil \frac{n}{4} \right\rceil \\ \frac{3n}{10} - 6 &\geq \frac{n}{4} + 1 \\ 12n - 240 &\geq 10n + 40 \\ n &\geq 140\end{aligned}$$

9.3-3

Show how quicksort can be made to run in $O(n \lg n)$ time in the worst case, assuming that all elements are distinct.

Use median as pivot, since we can find median in $O(n)$, and based on Problem 7-2 (b), we have $T(n) = T(n/2) + O(n)$.

9.3-4 *

Suppose that an algorithm uses only comparisons to find the i th smallest element in a set of n elements. Show that it can also find the $i - 1$ smaller elements and $n - i$ larger elements without performing additional comparisons.

...

9.3-5

Suppose that you have a "black-box" worst-case linear-time median subroutine. Give a simple, linear-time algorithm that solves the selection problem for an arbitrary order statistic.

```

def black_box_median(a, p, r):
    return sorted(a)[(p + r) // 2]

def partition(a, p, r, x):
    i = p - 1
    for k in range(p, r - 1):
        if a[k] == x:
            a[k], a[r - 1] = a[r - 1], a[k]
            break
    for k in range(p, r - 1):
        if a[k] < x:
            i += 1
            a[i], a[k] = a[k], a[i]
    i += 1
    a[i], a[r - 1] = a[r - 1], a[i]
    return i

def select(a, p, r, i):
    if p + 1 == r:
        return a[p]
    x = black_box_median(a, p, r)
    q = partition(a, p, r, x)
    k = q - p + 1
    if i == k:
        return a[q]
    if i < k:
        return select(a, p, q, i)
    return select(a, q + 1, r, i - k)

```

9.3-6

The k th **quantiles** of an n -element set are the $k - 1$ order statistics that divide the sorted set into k equal-sized sets (to within 1). Give an $O(n \lg k)$ -time algorithm to list the k th quantiles of a set.

Pre-calculate the positions of the quantiles in $O(k)$, we use the $O(n)$ select algorithm to find the $\lfloor k/2 \rfloor$ th position, after that the elements are divided into two sets by the pivot the $\lfloor k/2 \rfloor$ th position, we do it recursively in the two sets to find other positions. Since the maximum depth is $\lceil \lg k \rceil$, the total running time is $O(n \lg k)$.

```

def partition(a, p, r):
    x = a[r - 1]
    i = p - 1
    for k in range(p, r - 1):

```

```

        if a[k] < x:
            i += 1
            a[i], a[k] = a[k], a[i]
    i += 1
    a[i], a[r - 1] = a[r - 1], a[i]
    return i

def randomized_partition(a, p, r):
    x = random.randint(p, r - 1)
    a[x], a[r - 1] = a[r - 1], a[x]
    return partition(a, p, r)

def randomized_select(a, p, r, i):
    while True:
        if p + 1 == r:
            return p, a[p]
        q = randomized_partition(a, p, r)
        k = q - p + 1
        if i == k:
            return q, a[q]
        if i < k:
            r = q
        else:
            p = q + 1
            i -= k

def k_quantiles_sub(a, p, r, pos, f, e, quantiles):
    if f + 1 > e:
        return
    mid = (f + e) // 2
    q, val = randomized_select(a, p, r, pos[mid])
    quantiles[mid] = val
    k_quantiles_sub(a, p, q, pos, f, mid, quantiles)
    k = q - p + 1
    for i in xrange(mid + 1, e):
        pos[i] -= k
    k_quantiles_sub(a, q + 1, r, pos, mid + 1, e, quantiles)

def k_quantiles(a, k):
    num = len(a) / k
    mod = len(a) % k
    pos = [num for _ in xrange(k)]
    for i in xrange(mod):
        pos[i] += 1
    for i in xrange(1, k):
        pos[i] += pos[i - 1]
    quantiles = [0 for _ in xrange(k)]
    k_quantiles_sub(a, 0, len(a), pos, 0, len(pos), quantiles)
    return quantiles

```

9.3-7

Describe an $O(n)$ -time algorithm that, given a set S of n distinct numbers and a positive integer $k \leq n$, determines the k numbers in S that are closest to the median of S .

Find the median in $O(n)$; create a new array, each element is the absolute value of the original value subtract the median; find the k th smallest number in $O(n)$, then the desired values are the elements whose absolute difference with the median is less than or equal to the k th smallest number in the new array.

```
def black_box_kth(a, k):
    return sorted(a)[k-1]

def black_box_median(a):
    return sorted(a)[(len(a) - 1) // 2]

def k_closest(a, k):
    median = black_box_median(a)
    b = [abs(a[i] - median) for i in xrange(len(a))]
    kth = black_box_kth(b, k)
    closest = []
    for i in xrange(len(a)):
        if abs(a[i] - median) < kth:
            closest.append(a[i])
    for i in xrange(len(a)):
        if abs(a[i] - median) == kth:
            closest.append(a[i])
        if len(closest) >= k:
            break
    return closest
```

9.3-8

Let $X[1 \dots n]$ and $Y[1 \dots n]$ be two arrays, each containing n numbers already in sorted order. Give an $O(\lg n)$ -time algorithm to find the median of all $2n$ elements in arrays X and Y .

We can find the median in $O(1)$ time in a sorted array, compare the medians of the two array, if the median of X is less than the median of Y , then we know the median must located in the right side of X or left side of Y . Do it recursively, when there is only one element left in each array, the smaller one is the median.

```
def median_of_two(a, b):
    if len(a) == 1:
        return min(a[0], b[0])
    mid = (len(a) - 1) // 2
    k = mid + 1
    if a[mid] <= b[mid]:
        return median_of_two(a[-k:], b[:k])
    return median_of_two(a[:k], b[-k:])
```

9.3-9

Professor Olay is consulting for an oil company, which is planning a large pipeline running east to west through an oil field of n wells. The company wants to connect a spur pipeline from each well directly to the main pipeline along a shortest route (either north or south), as shown in Figure 9.2. Given the x - and y -coordinates of the wells, how should the professor pick the optimal location of the main pipeline, which would be the one that minimizes the total length of the spurs? Show how to determine the optimal location in linear time.

Find the median of y . Suppose n is odd, if we move the main pipeline slightly, then the total distance will be increased by $(n + 1)/2 \cdot d$ for one side and decreased by $(n - 1)/2 \cdot d$ for the other side, thus the total distance is increased by d .

Problems

9-1 Largest i numbers in sorted order

Given a set of n numbers, we wish to find the i largest in sorted order using a comparison-based algorithm. Find the algorithm that implements each of the following methods with the best asymptotic worst-case running time, and analyze the running times of the algorithms in terms of n and i .

a. Sort the numbers, and list the i largest.

Depends on the sorting algorithm, with heap sort the worst-case is $O(n \lg n + i)$.

b. Build a max-priority queue from the numbers, and call EXTRACT-MAX i times.

Build the heap is $O(n)$, extract is $O(\lg n)$, thus the worst time is $O(n + i \lg n)$.

c. Use an order-statistic algorithm to find the i th largest number, partition around that number, and sort the i largest numbers.

$$O(n + n + i \lg i) = O(n + i \lg i)$$

9-2 Weighted median

For n distinct elements x_1, x_2, \dots, x_n with positive weights w_1, w_2, \dots, w_n such that $\sum_{i=1}^n w_i = 1$, the **weighted (lower) median** is the element x_k satisfying

$$\sum_{x_i < x_k} w_i < \frac{1}{2}$$

and

$$\sum_{x_i > x_k} w_i \leq \frac{1}{2}$$

a. Argue that the median of x_1, x_2, \dots, x_n is the weighted median of the x_i with weights $w_i = 1/n$ for $i = 1, 2, \dots, n$.

$$\sum_{x_i < x_k} \frac{1}{n} < \frac{1}{2} \quad \sum_{x_i > x_k} \frac{1}{n} \leq \frac{1}{2}$$

$$\frac{m-1}{n} < \frac{1}{2} \quad \frac{n-m}{n} \geq \frac{1}{2}$$

$$m < \frac{n}{2} + 1 \quad m \geq \frac{n}{2}$$

$$\therefore \frac{n}{2} \leq m < \frac{n}{2} + 1$$

$$\therefore m = \left\lfloor \frac{n}{2} \right\rfloor$$

- b.** Show how to compute the weighted median of n elements in $O(n \lg n)$ worstcase time using sorting.

```
def weighted_median(x):
    x.sort()
    s = 0.0
    for i in range(len(x)):
        s += x[i]
        if s >= 0.5:
            return x[i]
```

- c.** Show how to compute the weighted median in $\Theta(n)$ worst-case time using a linear-time median algorithm such as SELECT from Section 9.3.

Use the median as pivot, the algorithm is exactly $T(n) = T(n/2) + \Theta(n) = \Theta(n)$.

```

def black_box_median(a, p, r):
    return sorted(a[p:r])[(r - p - 1) // 2]

def partition(a, p, r, x):
    s = x
    i = p - 1
    for j in xrange(p, r - 1):
        if a[j] == x:
            a[j], a[r - 1] = a[r - 1], a[j]
            break
    for j in xrange(p, r - 1):
        if a[j] < x:
            i += 1
            s += a[j]
            a[i], a[j] = a[j], a[i]
    i += 1
    a[i], a[r - 1] = a[r - 1], a[i]
    return i, s

def weighted_median(a, p, r, w=0.5):
    if p + 1 >= r:
        return a[p]
    x = black_box_median(a, p, r)
    q, s = partition(a, p, r, x)
    if s - a[q] < w and s >= w:
        return a[q]
    if s >= w:
        return weighted_median(a, p, q, w)
    return weighted_median(a, q + 1, r, w - s)

```

The **post-office location problem** is defined as follows. We are given n points p_1, p_2, \dots, p_n with associated weights w_1, w_2, \dots, w_n . We wish to find a point p that minimizes the sum $\sum_{i=1}^n w_i d(p, p_i)$, where $d(a, b)$ is the distance between points a and b .

d. Argue that the weighted median is a best solution for the 1-dimensional postoffice location problem, in which points are simply real numbers and the distance between points a and b is $d(a, b) = |a - b|$.

Same as Exercise 9.3-9.

- e. Find the best solution for the 2-dimensional post-office location problem, in which the points are (x, y) coordinate pairs and the distance between points $a = (x_1, y_1)$ and $b = (x_2, y_2)$ is the **Manhattan distance** given by

$$d(a, b) = |x_1 - x_2| + |y_1 - y_2|.$$

Since x and y are independent, the best solution is the medians of x and y separately.

9-3 Small order statistics

We showed that the worst-case number $T(n)$ of comparisons used by SELECT to select the i th order statistic from n numbers satisfies $T(n) = \Theta(n)$, but the constant hidden by the Θ -notation is rather large. When i is small relative to n , we can implement a different procedure that uses SELECT as a subroutine but makes fewer comparisons in the worst case.

- a. Describe an algorithm that uses $U_i(n)$ comparisons to find the i th smallest of n elements, where

$$U_i(n) = \begin{cases} T(n) & \text{if } i \geq n/2 \\ \lfloor n/2 \rfloor + U_i(\lceil n/2 \rceil) + T(2i) & \text{otherwise} \end{cases}$$

Divide elements into pairs and compare each pair. Recursively deal with the set with the smaller elements in each pair, and return the i smallest elements by partition, then the i th element belong to the pairs that appeared in the i smallest elements.

- b. Show that, if $i < n/2$, then $U_i(n) = n + O(T(2i) \lg(n/i))$.

Suppose $U_i(n) \leq n + cT(2i) \lg(n/i)$,

$$\begin{aligned} T(n) &= \lfloor n/2 \rfloor + U_i(\lceil n/2 \rceil) + T(2i) \\ &\leq n/2 + n/2 + cT(2i) \lg(n/2i) + T(2i) \\ &= n + cT(2i) \lg(n/i) + T(2i)(1 - c) \\ &\leq n + cT(2i) \lg(n/i) \quad (c \geq 1) \end{aligned}$$

- c. Show that if i is a constant less than $n/2$, then $U_i(n) = n + O(\lg n)$.

$$\begin{aligned}
U_i(n) &= n + O(T(2i) \lg(n/i)) \\
&= n + O(O(1)(\lg n - \lg i)) \\
&= n + O(\lg n - O(1)) \\
&= n + O(\lg n)
\end{aligned}$$

d. Show that if $i = n/k$ for $k \geq 2$, then $U_i(n) = n + O(T(2n/k) \lg k)$.

$$\begin{aligned}
U_i(n) &= n + O(T(2i) \lg(n/i)) \\
&= n + O(T(2n/k) \lg(k/2)) \\
&= n + O(T(2n/k) \lg k)
\end{aligned}$$

9-4 Alternative analysis of randomized selection

In this problem, we use indicator random variables to analyze the RANDOMIZED-SELECT procedure in a manner akin to our analysis of RANDOMIZED-QUICKSORT in Section 7.4.2.

As in the quicksort analysis, we assume that all elements are distinct, and we rename the elements of the input array A as z_1, z_2, \dots, z_n , where z_i is the i th smallest element. Thus, the call RANDOMIZED-SELECT $(A, 1, n, k)$ returns z_k .

For $1 \leq i < j \leq n$, let

$X_{ijk} = I\{z_i \text{ is compared with } z_j \text{ sometime during the execution of the algorithm to find } z_k\}$.

a. Give an exact expression for $E[X_{ijk}]$.

$$E[X_{ijk}] = \begin{cases} \frac{2}{j-k+1} & (k \leq i < j) \\ \frac{2}{j-i+1} & (i \leq k \leq j) \\ \frac{2}{k-i+1} & (i < j \leq k) \end{cases}$$

b. Let X_k denote the total number of comparisons between elements of array A when finding z_k . Show that

$$\begin{aligned}
 \mathbb{E}[X_k] &\leq 2 \left(\sum_{i=1}^k \sum_{j=k}^n \frac{1}{j-i+1} + \sum_{j=k+1}^n \frac{j-k-1}{j-k+1} + \sum_{i=1}^{k-2} \frac{k-i-1}{k-i+1} \right) \\
 \mathbb{E}[X_k] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbb{E}[X_{ijk}] \\
 &= \sum_{i=k+1}^{n-1} \sum_{j=i+1}^n \mathbb{E}[X_{ijk}] + \sum_{i=1}^k \sum_{j=k}^n \mathbb{E}[X_{ijk}] + \sum_{i=1}^{k-2} \sum_{j=i+1}^{k-1} \mathbb{E}[X_{ijk}] \\
 &= \sum_{i=k+1}^{n-1} \sum_{j=i+1}^n \frac{2}{j-k+1} + \sum_{i=1}^k \sum_{j=k}^n \frac{2}{j-i+1} + \sum_{i=1}^{k-2} \sum_{j=i+1}^{k-1} \frac{2}{k-i+1} \\
 &= 2 \cdot \left(\sum_{i=k+1}^{n-1} \sum_{j=i+1}^n \frac{1}{j-k+1} + \sum_{i=1}^k \sum_{j=k}^n \frac{1}{j-i+1} + \sum_{i=1}^{k-2} \sum_{j=i+1}^{k-1} \frac{1}{k-i+1} \right) \\
 &= 2 \left(\sum_{j=k+2}^{n-1} \sum_{i=k+1}^{j-1} \frac{1}{j-k+1} + \sum_{i=1}^k \sum_{j=k}^n \frac{1}{j-i+1} + \sum_{i=1}^{k-2} \sum_{j=i+1}^{k-1} \frac{1}{k-i+1} \right) \\
 &= 2 \left(\sum_{j=k+2}^{n-1} \frac{j-k-1}{j-k+1} + \sum_{i=1}^k \sum_{j=k}^n \frac{1}{j-i+1} + \sum_{i=1}^{k-2} \frac{k-i-1}{k-i+1} \right) \\
 &\leq 2 \left(\sum_{j=k+1}^n \frac{j-k-1}{j-k+1} + \sum_{i=1}^k \sum_{j=k}^n \frac{1}{j-i+1} + \sum_{i=1}^{k-2} \frac{k-i-1}{k-i+1} \right) \\
 &= 2 \left(\sum_{i=1}^k \sum_{j=k}^n \frac{1}{j-i+1} + \sum_{j=k+1}^n \frac{j-k-1}{j-k+1} + \sum_{i=1}^{k-2} \frac{k-i-1}{k-i+1} \right)
 \end{aligned}$$

c. Show that $\mathbb{E}[X_k] \leq 4n$.

Based on [StackExchange](#),

$$\sum_{i=1}^k \sum_{j=k}^n \frac{1}{j-i+1} \leq n$$

And

$$\sum_{j=k+1}^n \frac{j-k-1}{j-k+1} + \sum_{i=1}^{k-2} \frac{k-i-1}{k-i+1} \leq \sum_{j=k+1}^n 1 + \sum_{i=1}^{k-2} 1 = n - k + k - 2 = n - 2 < n$$

Therefore $E[X_k] \leq 4n$

d. Conclude that, assuming all elements of array A are distinct, RANDOMIZED-SELECT runs in expected time $O(n)$

$$O(4n) = O(n)$$

10 Elementary Data Structures

- 10.1 Stacks and queues
- 10.2 Linked lists
- 10.3 Implementing pointers and objects
- 10.4 Representing rooted trees
- Problems

10.1 Stacks and queues

10.1-1

Using Figure 10.1 as a model, illustrate the result of each operation in the sequence $\text{PUSH}(S, 4)$, $\text{PUSH}(S, 1)$, $\text{PUSH}(S, 3)$, $\text{POP}(S)$, $\text{PUSH}(S, 8)$, and $\text{POP}(S)$ on an initially empty stack S stored in array $S[1 \dots 6]$.

<https://github.com/CyberZHGU/>

$S.\text{top} = 1$



<https://github.com/CyberZHGU/>

$S.\text{top} = 2$



<https://github.com/CyberZHGU/>

$S.\text{top} = 3$



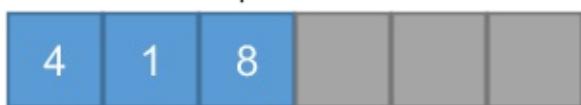
<https://github.com/CyberZHGU/>

$S.\text{top} = 2$



<https://github.com/CyberZHGU/>

$S.\text{top} = 3$



https://github.com/CyberZH/CS-Notes/blob/main/10.1-Stacks-and-queues/stacks/stack_1.png

S.top = 1



https://github.com/CyberZH/CS-Notes/blob/main/10.1-Stacks-and-queues/stacks/stack_2.png

S.top = 2



https://github.com/CyberZH/CS-Notes/blob/main/10.1-Stacks-and-queues/stacks/stack_3.png

S.top = 3



https://github.com/CyberZH/CS-Notes/blob/main/10.1-Stacks-and-queues/stacks/stack_4.png

S.top = 2



https://github.com/CyberZH/CS-Notes/blob/main/10.1-Stacks-and-queues/stacks/stack_5.png

S.top = 3



https://github.com/CyberZH/CS-Notes/blob/main/10.1-Stacks-and-queues/stacks/stack_6.png

S.top = 2



10.1-2

```

n = 100
a = [-1 for _ in xrange(n)]


class Stack1:
    def __init__(self):
        self.top = -1

    def is_empty(self):
        return self.top == -1

    def push(self, x):
        global a
        self.top += 1
        a[self.top] = x

    def pop(self):
        global a
        self.top -= 1
        return a[self.top + 1]


class Stack2:
    def __init__(self):
        self.top = n

    def is_empty(self):
        return self.top == n

    def push(self, x):
        global a
        self.top -= 1
        a[self.top] = x

    def pop(self):
        global a
        self.top += 1
        return a[self.top - 1]

```

10.1-3

Using Figure 10.2 as a model, illustrate the result of each operation in the sequence $\text{ENQUEUE}(Q, 4)$, $\text{ENQUEUE}(Q, 1)$, $\text{ENQUEUE}(Q, 3)$, $\text{DEQUEUE}(Q)$, $\text{ENQUEUE}(Q, 8)$, and $\text{DEQUEUE}(Q)$ on an initially empty queue Q stored in array $Q[1 \dots 6]$.

Using Figure 10.2 as a model, illustrate the result of each operation in the sequence $\text{ENQUEUE}(Q, 4)$, $\text{ENQUEUE}(Q, 1)$, $\text{ENQUEUE}(Q, 3)$, $\text{DEQUEUE}(Q)$, $\text{ENQUEUE}(Q, 8)$, and $\text{DEQUEUE}(Q)$ on an initially empty queue Q stored in array $Q[1 \dots 6]$.

<http://tiny.cc/meyarw1>

$Q.\text{head} = 1$



$Q.\text{tail} = 2$

<http://tiny.cc/meyarw1>

$Q.\text{head} = 1$



$Q.\text{tail} = 3$

<http://tiny.cc/meyarw1>

$Q.\text{head} = 1$



$Q.\text{tail} = 4$

<http://tiny.cc/meyarw1>

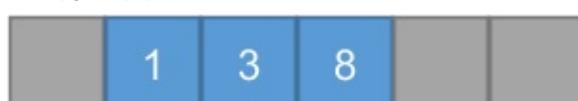
$Q.\text{head} = 2$



$Q.\text{tail} = 4$

<http://tiny.cc/meyarw1>

$Q.\text{head} = 2$



$Q.\text{tail} = 5$

<https://github.com/CyberZH/CS>

Q.head = 1



Q.tail = 2

<https://github.com/CyberZH/CS>

Q.head = 1



Q.tail = 3

<https://github.com/CyberZH/CS>

Q.head = 1



Q.tail = 4

<https://github.com/CyberZH/CS>

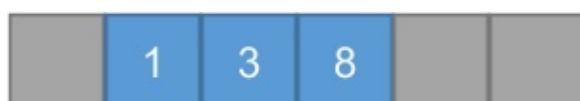
Q.head = 2



Q.tail = 4

<https://github.com/CyberZH/CS>

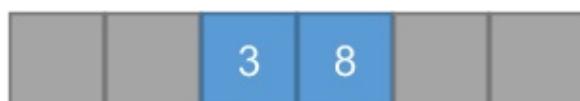
Q.head = 2



Q.tail = 5

<https://github.com/CyberZH/CS>

Q.head = 3



Q.tail = 5

```

class Deque:
    def __init__(self, size):
        self.q = [-1 for _ in xrange(size)]
        self.front = 0
        self.back = 0

    def push_front(self, x):
        if (self.back + 1) % len(self.q) == self.front:
            raise Exception('overflow')
        self.front -= 1
        if self.front == -1:
            self.front = len(self.q) - 1
        self.q[self.front] = x

    def push_back(self, x):
        if (self.back + 1) % len(self.q) == self.front:
            raise Exception('overflow')
        self.q[self.back] = x
        self.back += 1
        if self.back == len(self.q):
            self.back = 0

    def pop_front(self):
        if self.front == self.back:
            raise Exception('underflow')
        x = self.q[self.front]
        self.front += 1
        if self.front == len(self.q):
            self.front = 0
        return x

    def pop_back(self):
        if self.front == self.back:
            raise Exception('underflow')
        self.back -= 1
        if self.back == -1:
            self.back = len(self.q) - 1
        return self.q[self.back]

```

10.1-6

Show how to implement a queue using two stacks. Analyze the running time of the queue operations.

Enqueue: $\Theta(1)$

Dequeue: worst $O(n)$, amortized $\Theta(1)$

```

class Deque:
    def __init__(self, size):
        self.q = [-1 for _ in xrange(size)]
        self.front = 0
        self.back = 0

    def push_front(self, x):
        if (self.back + 1) % len(self.q) == self.front:
            raise Exception('overflow')
        self.front -= 1
        if self.front == -1:
            self.front = len(self.q) - 1
        self.q[self.front] = x

    def push_back(self, x):
        if (self.back + 1) % len(self.q) == self.front:
            raise Exception('overflow')
        self.q[self.back] = x
        self.back += 1
        if self.back == len(self.q):
            self.back = 0

    def pop_front(self):
        if self.front == self.back:
            raise Exception('underflow')
        x = self.q[self.front]
        self.front += 1
        if self.front == len(self.q):
            self.front = 0
        return x

    def pop_back(self):
        if self.front == self.back:
            raise Exception('underflow')
        self.back -= 1
        if self.back == -1:
            self.back = len(self.q) - 1
        return self.q[self.back]

```

10.1-6

Show how to implement a queue using two stacks. Analyze the running time of the queue operations.

Enqueue: $\Theta(1)$

Dequeue: worst $O(n)$, amortized $\Theta(1)$

```
class BlackBoxQueue:
    def __init__(self):
        self.s = []

    def is_empty(self):
        return len(self.s) == 0

    def enqueue(self, x):
        self.s.append(x)

    def dequeue(self):
        x = self.s[0]
        del self.s[0]
        return x


class Stack:
    def __init__(self):
        self.queue_in = BlackBoxQueue()
        self.queue_out = BlackBoxQueue()

    def is_empty(self):
        return self.queue_in.is_empty()

    def push(self, x):
        self.queue_in.enqueue(x)

    def pop(self):
        if self.queue_in.is_empty():
            raise Exception('underflow')
        while True:
            x = self.queue_in.dequeue()
            if self.queue_in.is_empty():
                break
            self.queue_out.enqueue(x)
        self.queue_in, self.queue_out = self.queue_out, self.queue_in
        return x
```

10.2 Linked lists

10.2-1

Can you implement the dynamic-set operation INSERT on a singly linked list in $O(1)$ time? How about DELETE?

INSERT: $O(1)$

DELETE: $O(n)$

```
class LinkListNode:
    def __init__(self, value):
        self.value = value
        self.next = None

    def to_str(head):
        values = []
        head = head.next
        while head is not None:
            values.append(head.value)
            head = head.next
        return ' '.join(map(str, values))

    def insert(head, x):
        new_node = LinkListNode(x)
        new_node.next = head.next
        head.next = new_node

    def delete(head, x):
        while head is not None:
            if head.next is not None and head.next.value == x:
                head.next = head.next.next
            else:
                head = head.next
```

10.2-2

Implement a stack using a singly linked list L . The operations PUSH and POP should still take $O(1)$ time.

```
class LinkListNode:  
    def __init__(self, value):  
        self.value = value  
        self.next = None  
  
def push(head, x):  
    new_node = LinkListNode(x)  
    new_node.next = head.next  
    head.next = new_node  
  
def pop(head):  
    if head.next is None:  
        return None  
    x = head.next.value  
    head.next = head.next.next  
    return x
```

10.2-3

Implement a queue by a singly linked list L . The operations ENQUEUE and DEQUEUE should still take $O(1)$ time.

```

class LinkListNode:
    def __init__(self, value):
        self.value = value
        self.next = None

class Queue:
    def __init__(self):
        self.head = None
        self.tail = LinkListNode(None)

    def enqueue(self, x):
        new_node = LinkListNode(x)
        if self.tail.next is None:
            self.head = new_node
            self.tail.next = self.head
        else:
            self.head.next = new_node
            self.head = new_node

    def dequeue(self):
        if self.tail.next is None:
            return None
        x = self.tail.next.value
        self.tail = self.tail.next
        return x

```

10.2-4

As written, each loop iteration in the LIST-SEARCH' procedure requires two tests: one for $x \neq L.nil$ and one for $x.key \neq k$. Show how to eliminate the test for $x \neq L.nil$ in each iteration.

```

class LinkListNode:
    def __init__(self, key):
        self.key = key
        self.next = None
        self.prev = None

class LinkList:
    def __init__(self):
        self.nil = LinkListNode(None)
        self.nil.next = self.nil
        self.nil.prev = self.nil

    def insert(self, x):
        x = LinkListNode(x)
        x.next = self.nil.next
        x.prev = self.nil
        x.next.prev = x
        x.prev.next = x

    def search(self, k):
        self.nil.key = k
        x = self.nil.next
        while x.key != k:
            x = x.next
        if x == self.nil:
            return None
        return x

```

10.2-5

Implement the dictionary operations INSERT, DELETE, and SEARCH using singly linked, circular lists. What are the running times of your procedures?

$\text{INSERT } \Theta(n)$, $\text{DELETE } \Theta(n)$, $\text{SEARCH } \Theta(n)$.

```

class LinkListNode:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.next = None
        self.prev = None

class Dict:
    def __init__(self):
        self.nil = LinkListNode(None, None)
        self.nil.next = self.nil
        self.nil.prev = self.nil

    def insert(self, key, value):
        x = self.search_node(key)
        if x is None:
            x = LinkListNode(key, value)
            x.next = self.nil.next
            x.prev = self.nil
            x.next.prev = x
            x.prev.next = x
        else:
            x.value = value

    def delete(self, key):
        x = self.search_node(key)
        if x is not None:
            x.next.prev = x.prev
            x.prev.next = x.next

    def search_node(self, key):
        self.nil.key = key
        x = self.nil.next
        while x.key != key:
            x = x.next
        if x == self.nil:
            return None
        return x

    def search(self, key):
        x = self.search_node(key)
        if x is None:
            return None
        return x.value

```

10.2-6

The dynamic-set operation UNION takes two disjoint sets S_1 and S_2 as input, and it returns a set $S = S_1 \cup S_2$ consisting of all the elements of S_1 and S_2 . The sets S_1 and S_2 are usually destroyed by the operation. Show how to support UNION in $O(1)$ time using a suitable list data structure.

```

class LinkListNode:
    def __init__(self, key):
        self.key = key
        self.next = None
        self.prev = None


class LinkList:
    def __init__(self):
        self.nil = LinkListNode(None)
        self.nil.next = self.nil
        self.nil.prev = self.nil

    def insert(self, key):
        x = LinkListNode(key)
        x.next = self.nil.next
        x.prev = self.nil
        x.next.prev = x
        x.prev.next = x

    def values(self):
        values = []
        x = self.nil.next
        while x != self.nil:
            values.append(x.key)
            x = x.next
        return values

    def union(self, list_2):
        list_1.nil.next.prev = list_2.nil.prev
        list_2.nil.prev.next = list_1.nil.next
        list_1.nil.next = list_2.nil.next
        list_2.nil.next.prev = list_1.nil
        return list_1

```

10.2-7

Give a $\Theta(n)$ -time nonrecursive procedure that reverses a singly linked list of n elements. The procedure should use no more than constant storage beyond that needed for the list itself.

```

class LinkListNode:
    def __init__(self, value):
        self.value = value
        self.next = None

def to_list(head):
    values = []
    head = head.next
    while head is not None:
        values.append(head.value)
        head = head.next
    return values

def insert(head, x):
    new_node = LinkListNode(x)
    new_node.next = head.next
    head.next = new_node

def reverse(head):
    prev = None
    node = head.next
    while node is not None:
        next_node = node.next
        node.next = prev
        prev = node
        node = next_node
    head.next = prev

```

10.2-8 *

Explain how to implement doubly linked lists using only one pointer value $x.\text{np}$ per item instead of the usual two (next and prev). Assume all pointer values can be interpreted as k -bit integers, and define $x.\text{np}$ to be $x.\text{np} = x.\text{next} \text{ XOR } x.\text{prev}$, the k -bit "exclusive-or" of $x.\text{next}$ and $x.\text{prev}$. (The value NIL is represented by 0.) Be sure to describe what information you need to access the head of the list. Show how to implement the SEARCH, INSERT, and DELETE operations on such a list. Also show how to reverse such a list in $O(1)$ time.

$$head.\text{np} = \text{next}$$

$$tail.\text{np} = \text{prev}$$

$$\text{next} = x.\text{np} \text{ XOR } \text{prev}$$

$$\text{prev} = x.\text{np} \text{ XOR } \text{next}$$

Reverse:

$$\text{head. } np. \text{ } np = (\text{head} \text{ } \texttt{XOR} \text{ } \text{head. } np. \text{ } np) \text{ } \texttt{XOR} \text{ } tail$$

$$\text{tail. } np. \text{ } np = (\text{tail} \text{ } \texttt{XOR} \text{ } \text{tail. } np. \text{ } np) \text{ } \texttt{XOR} \text{ } head$$

$$\text{head. } np, \text{tail. } np = \text{tail. } np, \text{head. } np$$

10.3 Implementing pointers and objects

10.3-1

Draw a picture of the sequence $\langle 13; 4; 8; 19; 5; 11 \rangle$ stored as a doubly linked list using the multiple-array representation. Do the same for the single-array representation.

...

10.3-2

Write the procedures ALLOCATE-OBJECT and FREE-OBJECT for a homogeneous collection of objects implemented by the single-array representation.

...

10.3-3

Why don't we need to set or reset the prev attributes of objects in the implementation of the ALLOCATE-OBJECT and FREE-OBJECT procedures?

Because we do not need to know prev.

10.3-4

It is often desirable to keep all elements of a doubly linked list compact in storage, using, for example, the first m index locations in the multiple-array representation. (This is the case in a paged, virtual-memory computing environment.) Explain how to implement the procedures ALLOCATE-OBJECT and FREE-OBJECT so that the representation is compact. Assume that there are no pointers to elements of the linked list outside the list itself.

See 10.3-5.

10.3-5

Let L be a doubly linked list of length n stored in arrays key , $prev$, and $next$ of length m . Suppose that these arrays are managed by ALLOCATE-OBJECT and FREE-OBJECT procedures that keep a doubly linked free list F . Suppose further that of the m items, exactly n are on list L and $m - n$ are on the free list. Write a procedure COMPACTIFY-LIST (L, F) that, given the list L and the free list F , moves the items in L so that they occupy array positions $1, 2, \dots, n$ and adjusts the free list F so that it remains correct, occupying array positions $n + 1, n + 2, \dots, m$. The running time of your procedure should be $\Theta(n)$, and it should use only a constant amount of extra space. Argue that your procedure is correct.

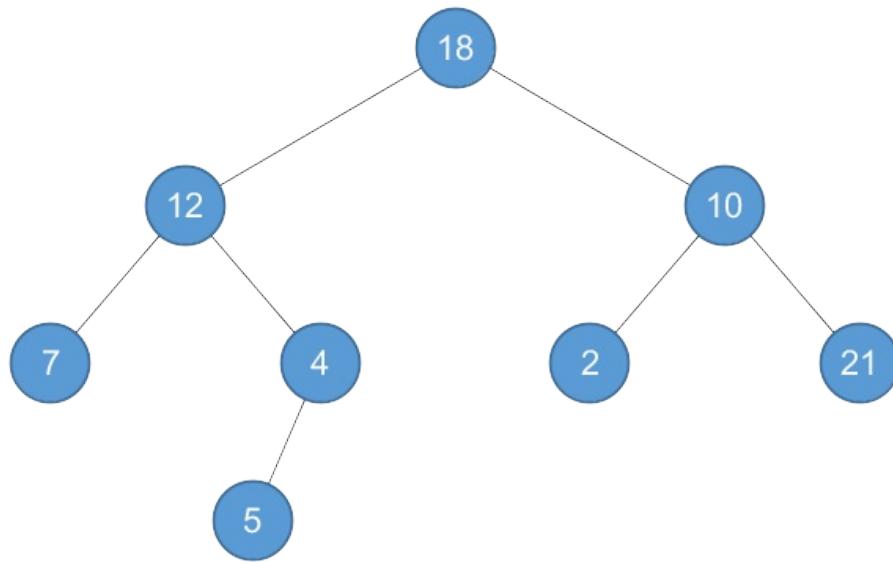
For the i th element, if it is not in position i and position i is not free, we move the element at position i to a new allocated position, and move the i th element to position i .

10.4 Representing rooted trees

10.4-1

Draw the binary tree rooted at index 6 that is represented by the following attributes:

index	key	left	right
1	12	7	3
2	15	8	NIL
3	4	10	NIL
4	10	5	9
5	2	NIL	NIL
6	18	1	4
7	7	NIL	NIL
8	14	6	2
9	21	NIL	NIL
10	5	NIL	NIL

**10.4-2**

Write an $O(n)$ -time recursive procedure that, given an n -node binary tree, prints out the key of each node in the tree.

```

class TreeNode:
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

    def print_tree(node):
        if node is not None:
            print(node.value)
            print_tree(node.left)
            print_tree(node.right)
  
```

10.4-3

Write an $O(n)$ -time nonrecursive procedure that, given an n -node binary tree, prints out the key of each node in the tree. Use a stack as an auxiliary data structure.

```
class TreeNode:
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

    def print_tree(node):
        stack = [node]
        while len(stack) > 0:
            node = stack[-1]
            del stack[-1]
            if node is not None:
                print(node.value)
                stack.append(node.left)
                stack.append(node.right)
```

10.4-4

Write an $O(n)$ -time procedure that prints all the keys of an arbitrary rooted tree with n nodes, where the tree is stored using the left-child, right-sibling representation.

```
class TreeNode:
    def __init__(self, value, parent=None, left=None, right=None):
        self.value = value
        self.parent = parent
        self.left_child = left
        self.right_sibling = right

    def print_tree(node):
        if node is not None:
            while node.parent is not None:
                node = node.parent
            while node is not None:
                print(node.value)
                sibling = node.right_sibling
                while sibling is not None:
                    print(sibling.value)
                    sibling = sibling.right_sibling
                node = node.left_child
```

10.4-5 ★

Write an $O(n)$ -time nonrecursive procedure that, given an n -node binary tree, prints out the key of each node. Use no more than constant extra space outside of the tree itself and do not modify the tree, even temporarily, during the procedure.

```
class TreeNode:
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.parent = None
        self.left = left
        self.right = right
        if left is not None:
            left.parent = self
        if right is not None:
            right.parent = self

def print_tree(node):
    prev = None
    while node is not None:
        if node.parent == prev:
            print(node.value)
            prev = node
        if node.left is None:
            node = node.parent
        else:
            node = node.left
        elif node.left == prev:
            prev = node
            if node.right is None:
                node = node.parent
            else:
                node = node.right
        else:
            prev = node
            node = node.parent
```

10.4-6 *

The left-child, right-sibling representation of an arbitrary rooted tree uses three pointers in each node: *left-child*, *right-sibling*, and *parent*. From any node, its parent can be reached and identified in constant time and all its children can be reached and identified in time linear in the number of children. Show how to use only two pointers and one boolean value in each node so that the parent of a node or all of its children can be reached and identified in time linear in the number of children.

Use boolean to identify the last sibling, and the last sibling's right-sibling points to the parent.

Problems

10-1 Comparisons among lists

For each of the four types of lists in the following table, what is the asymptotic worst-case running time for each dynamic-set operation listed?

	unsorted, singly linked	sorted, singly linked	unsorted, doubly linked	sorted, doubly linked
SEARCH (L, k)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
INSERT (L, x)	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
DELETE (L, x)	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
SUCCESSOR (L, x)	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
PREDECESSOR (L, x)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
MINIMUM (L)	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
MAXIMUM (L)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

10-2 Mergeable heaps using linked lists

A mergeable heap supports the following operations: MAKE-HEAP (which creates an empty mergeable heap), INSERT, MINIMUM, EXTRACT-MIN, and UNION. Show how to implement mergeable heaps using linked lists in each of the following cases. Try to make each operation as efficient as possible. Analyze the running time of each operation in terms of the size of the dynamic set(s) being operated on.

a. Lists are sorted.

MAKE-HEAP $\Theta(1)$, INSERT $\Theta(n)$, MINIMUM $\Theta(1)$, EXTRACT-MIN $\Theta(1)$, UNION $\Theta(n)$

```
class LinkedListNode:
    def __init__(self, value):
        self.value = value
```

```

        self.next = None

class MergeableHeap:
    def __init__(self):
        self.head = None

    def to_list(self):
        values = []
        x = self.head
        while x is not None:
            values.append(x.value)
            x = x.next
        return values

    def insert(self, value):
        new_node = LinkedListNode(value)
        if self.head is None:
            self.head = new_node
        else:
            if value < self.head.value:
                new_node.next = self.head
                self.head = new_node
            else:
                x = self.head
                while x.next is not None and x.next.value < value:
                    x = x.next
                if x.next is None or x.next < value:
                    new_node.next = x.next
                    x.next = new_node

    def minimum(self):
        if self.head is None:
            return None
        return self.head.value

    def extract_min(self):
        if self.head is None:
            return None
        x = self.head.value
        self.head = self.head.next
        return x

    def union(self, other):
        head = LinkedListNode(None)
        x = head
        while self.head is not None or other.head is not None:
            if other.head is None:
                x.next = self.head
                self.head = self.head.next
            elif self.head is None:
                x.next = other.head
                other.head = other.head.next
            else:
                if self.head.value < other.head.value:
                    x.next = self.head
                    self.head = self.head.next
                else:
                    x.next = other.head
                    other.head = other.head.next
        return head

```

```

        elif self.head.value <= other.head.value:
            x.next = self.head
            self.head = self.head.next
        else:
            x.next = other.head
            other.head = other.head.next
        if x.next.value != x.value:
            x = x.next
    x.next = None
    self.head = head.next

```

b. Lists are unsorted.

MAKE-HEAP $\Theta(1)$, INSERT $\Theta(1)$, MINIMUM $\Theta(n)$, EXTRACT-MIN $\Theta(n)$, UNION $\Theta(n)$

```

class LinkedListNode:
    def __init__(self, value):
        self.value = value
        self.next = None

class MergeableHeap:
    def __init__(self):
        self.head = None

    def to_list(self):
        values = []
        x = self.head
        while x is not None:
            values.append(x.value)
            x = x.next
        return values

    def insert(self, value):
        x = LinkedListNode(value)
        if self.head is None:
            self.head = x
        else:
            x.next = self.head
            self.head = x

    def minimum(self):
        if self.head is None:
            return None
        min_val = self.head.value
        x = self.head.next
        while x is not None:
            min_val = min(min_val, x.value)
            x = x.next

```

```
return min_val

def delete(self, value):
    prev = None
    x = self.head
    while x is not None:
        if x.value == value:
            if x == self.head:
                self.head = self.head.next
            prev.next = x.next
            prev = x
            x = x.next

    def extract_min(self):
        x = self.minimum()
        self.delete(x)
        return x

def union(self, other):
    if self.head is None:
        self.head = other.head
    else:
        x = self.head
        while x.next is not None:
            x = x.next
        x.next = other.head
```

c. Lists are unsorted, and dynamic sets to be merged are disjoint.

Same as **b**.

10-3 Searching a sorted compact list

- a. Suppose that COMPACT-LIST-SEARCH (L, n, k) takes t iterations of the while loop of lines 2–8. Argue that COMPACT-LIST-SEARCH' (L, n, k, t) returns the same answer and that the total number of iterations of both the for and while loops within COMPACT-LIST-SEARCH' is at least t .
- b. Argue that the expected running time of COMPACT-LIST-SEARCH' (L, n, k, t) is $O(t + \mathbb{E}[X_t])$.
- c. Show that $\mathbb{E}[X_t] \leq \sum_{r=1}^n (1 - r/n)^t$.
- d. Show that $\sum_{r=0}^{n-1} r^t \leq n^{t+1}/(t+1)$.
- e. Prove that $\mathbb{E}[X_t] \leq n/(t+1)$.
- f. Show that COMPACT-LIST-SEARCH' (L, n, k, t) runs in $O(t + n/t)$ expected time.
- g. Conclude that COMPACT-LIST-SEARCH runs in $O(\sqrt{n})$ expected time.
- h. Why do we assume that all keys are distinct in COMPACT-LIST-SEARCH? Argue that random skips do not necessarily help asymptotically when the list contains repeated key values.

11 Hash Tables

- 11.1 Direct-address tables
- 11.2 Hash tables
- 11.3 Hash functions
- 11.4 Open addressing
- 11.5 Perfect hashing
- Problems

11.1 Direct-address tables

11.1-1

Suppose that a dynamic set S is represented by a direct-address table T of length m . Describe a procedure that finds the maximum element of S . What is the worst-case performance of your procedure?

Traverse the table, $O(m)$.

11.1-2

A **bit vector** is simply an array of bits (0s and 1s). A bit vector of length m takes much less space than an array of m pointers. Describe how to use a bit vector to represent a dynamic set of distinct elements with no satellite data. Dictionary operations should run in $O(1)$ time.

Set the corresponding bit to 0 or 1.

11.1-3

Suggest how to implement a direct-address table in which the keys of stored elements do not need to be distinct and the elements can have satellite data. All three dictionary operations (INSERT, DELETE, and SEARCH) should run in $O(1)$ time. (Don't forget that DELETE takes as an argument a pointer to an object to be deleted, not a key.)

Each key contains a linked list.

11.1-4 *

We wish to implement a dictionary by using direct addressing on a huge array. At the start, the array entries may contain garbage, and initializing the entire array is impractical because of its size. Describe a scheme for implementing a direct-address dictionary on a huge array. Each stored object should use $O(1)$ space; the operations SEARCH, INSERT, and DELETE should take $O(1)$ time each; and initializing the data structure should take $O(1)$ time.

Use an additional array, treated somewhat like a stack whose size is the number of keys actually stored in the dictionary. When INSERT, the value in the huge array is set to the top index of the additional array, and the additional array records the corresponding index in the huge array (and the satellite data). When DELETE, set the value in the huge array to -1.

```
class Item:
    def __init__(self):
        self.key = id(self) // 64 % 10007
        self.value = id(self)

huge_array = [random.randint(0, 10000) for _ in range(10007)]
additional_array = []

def insert(x):
    global huge_array
    global additional_array
    huge_array[x.key] = len(additional_array)
    additional_array.append((x.key, x))

def delete(x):
    global huge_array
    huge_array[x.key] = -1

def search(k):
    global huge_array
    global additional_array
    idx = huge_array[k]
    if 0 <= idx < len(additional_array):
        if additional_array[idx][0] == k:
            return additional_array[idx][1]
    return None
```

11.2 Hash tables

11.2-1

Suppose we use a hash function h to hash n distinct keys into an array T of length m . Assuming simple uniform hashing, what is the expected number of collisions? More precisely, what is the expected cardinality of $\{\{k, l\} : k \neq l \text{ and } h(k) = h(l)\}$?

$$\Pr\{h(k_i) = h(k_j)\} = 1/m$$

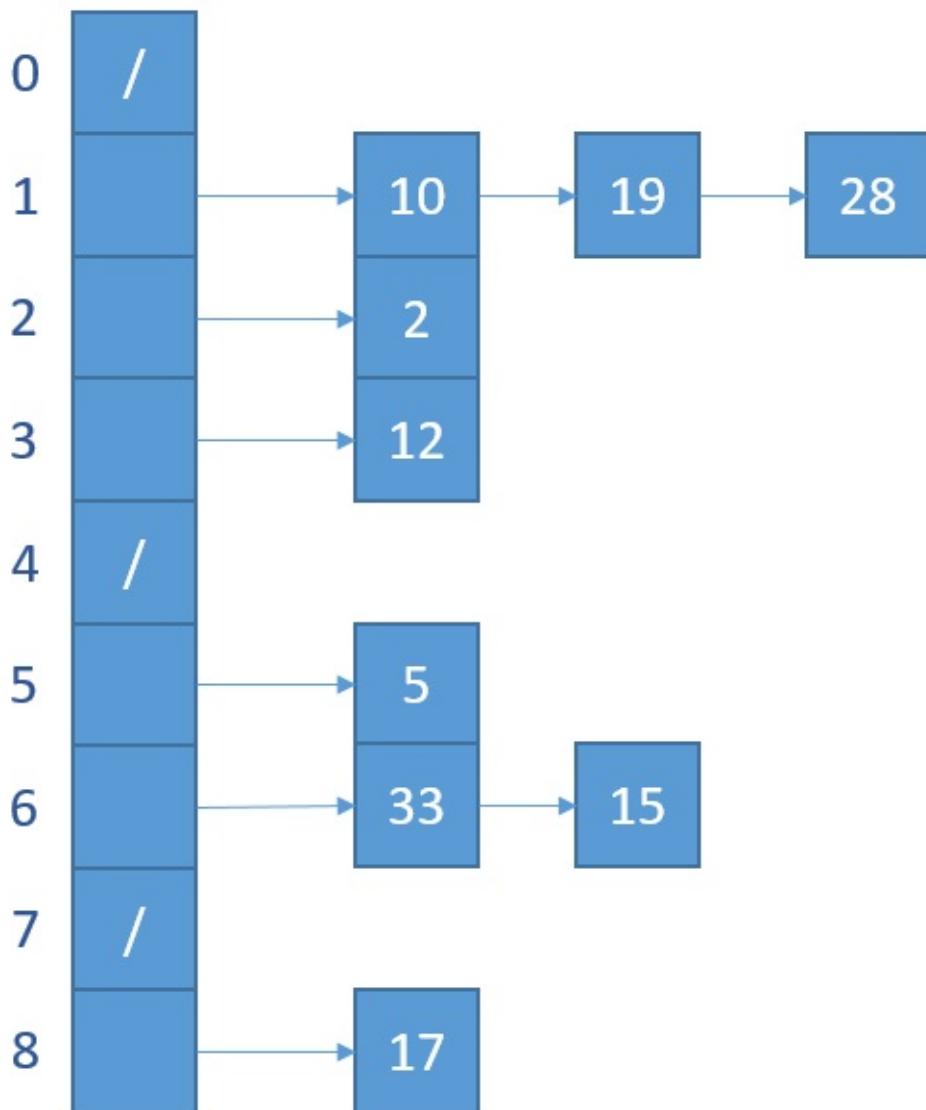
$$X_{ij} = \mathbf{I}\{h(k_i) = h(k_j)\}$$

$$\mathbf{E}[X_{ij}] = 1/m$$

$$\begin{aligned} \mathbf{E}[\{\{k, l\} : k \neq l \text{ and } h(k) = h(l)\}] &= \sum_{i=1}^n \sum_{j=i+1}^n \mathbf{E}[X_{ij}] \\ &= \sum_{i=1}^n \sum_{j=i+1}^n \frac{1}{m} \\ &= \frac{n(n-1)}{2m} \end{aligned}$$

11.2-2

Demonstrate what happens when we insert the keys 5, 28, 19, 15, 20, 33, 12, 17, 10 into a hash table with collisions resolved by chaining. Let the table have 9 slots, and let the hash function be $h(k) = k \bmod 9$.



11.2-3

Professor Marley hypothesizes that he can obtain substantial performance gains by modifying the chaining scheme to keep each list in sorted order. How does the professor's modification affect the running time for successful searches, unsuccessful searches, insertions, and deletions?

Successful searches: no difference, $\Theta(1 + \alpha)$

Unsuccessful searches: faster but still $\Theta(1 + \alpha)$

Insertions: same as successful searches, $\Theta(1 + \alpha)$

Deletions: same as successful searches, $\Theta(1 + \alpha)$

11.2-4

Suggest how to allocate and deallocate storage for elements within the hash table itself by linking all unused slots into a free list. Assume that one slot can store a flag and either one element plus a pointer or two pointers. All dictionary and free-list operations should run in $O(1)$ expected time. Does the free list need to be doubly linked, or does a singly linked free list suffice?

Flag: free or used.

If the slot is free, it contains two pointers point to the previous and the next free slots.

If the slot is used, it contains an element the the pointer to the next element with the same key.

We have to use a doubly linked list since we need $\Theta(1)$ deletion.

11.2-5

Suppose that we are storing a set of n keys into a hash table of size m . Show that if the keys are drawn from a universe U with $|U| > nm$, then U has a subset of size n consisting of keys that all hash to the same slot, so that the worst-case searching time for hashing with chaining is $\Theta(n)$.

Suppose the $m - 1$ slots contains at most $n - 1$ elements, then the remaining slot should have $|U| - (m - 1)(n - 1) > nm - (m - 1)(n - 1) = n + m - 1 \geq n$ elements, thus U has a subset of size n .

11.2-6

Suppose we have stored n keys in a hash table of size m , with collisions resolved by chaining, and that we know the length of each chain, including the length L of the longest chain. Describe a procedure that selects a key uniformly at random from among the keys in the hash table and returns it in expected time $O(L \cdot (1 + 1/\alpha))$.

Select a random key k and select a random number l such that $1 \leq l \leq L$. If $l \leq n_k$, the selected element is returned, otherwise repeat the procedure until we find an existing element.

$$\Pr\{\text{the selected element exists}\} = \alpha/L$$

$$E[\text{number of attempts}] = L/\alpha$$

When we find an existing element, we need $O(L)$ time to iterate to the element, thus the expected time is:

$$O(L/\alpha + L) = O(L \cdot (1 + 1/\alpha))$$

11.3 Hash functions

11.3-1

Suppose we wish to search a linked list of length n , where each element contains a key k along with a hash value $h(k)$. Each key is a long character string. How might we take advantage of the hash values when searching the list for an element with a given key?

Compare the long character strings only when they have the same hash values.

11.3-2

Suppose that we hash a string of r characters into m slots by treating it as a radix-128 number and then using the division method. We can easily represent the number m as a 32-bit computer word, but the string of r characters, treated as a radix-128 number, takes many words. How can we apply the division method to compute the hash value of the character string without using more than a constant number of words of storage outside the string itself?

We should calculate

$$\sum_{i=0}^{r-1} c_i \cdot 128^i \mod m$$

It cannot be calculated with a constant number of words of storage because the sum may exceed $2^{32} - 1$. However, Equation 31.18 suggests

$$\begin{aligned} \sum_{i=0}^{r-1} c_i \cdot 128^i &\equiv \sum_{i=0}^{r-1} (c_i \cdot 128^i) \mod m \pmod{m} \\ &\equiv \sum_{i=0}^{r-1} (c_i \cdot 128^i \mod m) \pmod{m} \\ &\equiv \sum_{i=1}^{r-1} (c_i \cdot 128^i \mod m) + c_0 \mod m \pmod{m} \\ &\equiv \sum_{i=1}^{r-1} ((c_i \cdot 128^{i-1} \mod m) + c_1 \mod m) \mod m \cdot (128 \mod m) \mod m + c_0 \mod m \pmod{m} \\ &\equiv \dots \\ &\equiv (\dots (c_{r-1} \mod m \cdot (128 \mod m) \mod m + c_{r-2} \mod m) \mod m \dots \cdot (128 \mod m) + c_1 \mod m) \\ &\quad \mod m + c_0 \mod m \pmod{m} \end{aligned}$$

It can be calculated with a loop.

```

sum := 0
for i = 1 to r
    sum := ((sum % m) * (128 % m) % m + s[i] % m) % m

```

And it fits in a word now. Furthermore, we may apply Equation 31.18 again and get

```

sum := 0
for i = 1 to r
    sum := (sum * 128 + s[i]) % m

```

Use `sum` as the key.

11.3-3

Consider a version of the division method in which $h(k) = k \bmod m$, where $m = 2^p - 1$ and k is a character string interpreted in radix 2^p . Show that if we can derive string x from string y by permuting its characters, then x and y hash to the same value. Give an example of an application in which this property would be undesirable in a hash function.

$$2^p \bmod (2^p - 1) = 1$$

$$c \cdot (2^p)^x \bmod (2^p - 1) = c \cdot 1^x = c$$

Thus the hashing is equivalent to $(\sum c_i) \bmod m$, the strings with different permutations will have the same hashing value.

11.3-4

Consider a hash table of size $m = 1000$ and a corresponding hash function $h(k) = \lfloor m(kA \bmod 1) \rfloor$ for $A = (\sqrt{5} - 1)/2$. Compute the locations to which the keys 61, 62, 63, 64, and 65 are mapped.

$$h(61) = 700$$

$$h(62) = 318$$

$$h(63) = 936$$

$$h(64) = 554$$

$$h(65) = 172$$

11.3-5 *

Define a family \mathcal{H} of hash functions from a finite set U to a finite set B to be ϵ -universal if for all pairs of distinct elements k and l in U ,

$$\Pr\{h(k) = h(l)\} \leq \epsilon,$$

where the probability is over the choice of the hash function h drawn at random from the family \mathcal{H} . Show that an ϵ -universal family of hash functions must have

$$\epsilon \geq \frac{1}{|B|} - \frac{1}{|U|}.$$

Suppose n_i is the number of elements in slot i , then the total number of collisions is:

Suppose $|U| = n$ and $|B| = m$

$$\begin{aligned} \sum_{i=1}^m \frac{n_i(n_i - 1)}{2} &= \frac{1}{2} \sum_{i=1}^m n_i^2 - \frac{1}{2} \sum_{i=1}^m n_i \\ &\geq \frac{1}{2} \sum_{i=1}^m \left(\frac{n}{m}\right)^2 - \frac{n}{2} \\ &= \frac{n^2}{2m} - \frac{n}{2} \end{aligned}$$

$$\epsilon \geq \frac{\sum_{i=1}^m \frac{n_i(n_i - 1)}{2}}{\frac{n(n - 1)}{2}}$$

$$\begin{aligned} &= \frac{\frac{n^2}{m} - n}{n(n - 1)} \\ &= \frac{n - m}{m(n - 1)} \\ &= \frac{n}{m(n - 1)} - \frac{1}{n - 1} \\ &\geq \frac{n}{mn} - \frac{1}{n} \\ &= \frac{1}{m} - \frac{1}{n} \end{aligned}$$

$$\epsilon \geq \frac{1}{|B|} - \frac{1}{|U|}$$

Therefore

11.3-6 *

Let U be the set of n -tuples of values drawn from \mathbb{Z}_p , and let $B = \mathbb{Z}_p$, where p is prime. Define the hash function $h_b : U \rightarrow B$ for $b \in \mathbb{Z}_p$ on an input n -tuple $\langle a_0, a_1, \dots, a_{n-1} \rangle$ from U as

$$h_b(\langle a_0, a_1, \dots, a_{n-1} \rangle) = \left(\sum_{j=0}^{n-1} a_j b^j \right) \bmod p,$$

and let $\mathcal{H} = \{h_b : b \in \mathbb{Z}_p\}$. Argue that \mathcal{H} is $((n-1)/p)$ -universal according to the definition of ϵ -universal in Exercise 11.3-5.

...

11.4 Open addressing

11.4-1

Consider inserting the keys $10, 22, 31, 4, 15, 28, 17, 88, 59$ into a hash table of length $m = 11$ using open addressing with the auxiliary hash function $h'(k) = k$. Illustrate the result of inserting these keys using linear probing, using quadratic probing with $c_1 = 1$ and $c_2 = 3$, and using double hashing with $h_1(k) = k$ and $h_2(k) = 1 + (k \bmod (m - 1))$.

```

m = 11

class LinearProbing:
    def __init__(self):
        global m
        self.slots = [None for _ in xrange(m)]

    def insert(self, key):
        global m
        i = 0
        while True:
            pos = (key + i) % m
            if self.slots[pos] is None:
                break
            i += 1
        self.slots[pos] = key


class QuadraticProbing:
    def __init__(self):
        global m
        self.slots = [None for _ in xrange(m)]

    def insert(self, key):
        global m
        i = 0
        while True:
            pos = (key + i + 3 * i * i) % m
            if self.slots[pos] is None:
                break
            i += 1
        self.slots[pos] = key


class DoubleHashing:
    def __init__(self):
        global m
        self.slots = [None for _ in xrange(m)]

    def insert(self, key):
        global m
        i = 0
        h2 = 1 + (key % (m - 1))
        while True:
            pos = (key + i * h2) % m
            if self.slots[pos] is None:
                break
            i += 1
        self.slots[pos] = key

```

Linear: [22, 88, None, None, 4, 15, 28, 17, 59, 31, 10]

Quadratic: [22, None, 88, 17, 4, None, 28, 59, 15, 31, 10]

Double: [22, None, 59, 17, 4, 15, 28, 88, None, 31, 10]

11.4-2

Write pseudocode for HASH-DELETE as outlined in the text, and modify HASHINSERT to handle the special value DELETED.

```
m = 5

class LinearProbing:
    def __init__(self):
        global m
        self.slots = [None for _ in xrange(m)]

    def insert(self, key):
        global m
        i = 0
        while True:
            pos = (key + i) % m
            if self.slots[pos] is None or self.slots[pos] == '[Deleted]':
                break
            i += 1
        self.slots[pos] = key

    def delete(self, key):
        global m
        i = 0
        while True:
            pos = (key + i) % m
            if self.slots[pos] is None:
                break
            if self.slots[pos] == key:
                self.slots[pos] = '[Deleted]'
                break
            i += 1
```

11.4-3

Consider an open-address hash table with uniform hashing. Give upper bounds on the expected number of probes in an unsuccessful search and on the expected number of probes in a successful search when the load factor is $3/4$ and when it is $7/8$.

$$\alpha = 3/4, \text{ unsuccessful: } \frac{1}{1 - \frac{3}{4}} = 4 \quad \text{probes, successful: } \frac{\frac{1}{3} \ln \frac{1}{1 - \frac{3}{4}}}{\frac{1}{4}} \approx 1.848 \text{ probes.}$$

$$\alpha = 7/8, \text{ unsuccessful: } \frac{1}{1 - \frac{7}{8}} = 8 \quad \text{probes, successful: } \frac{\frac{1}{7} \ln \frac{1}{1 - \frac{7}{8}}}{\frac{1}{8}} \approx 2.377 \text{ probes.}$$

11.4-4 *

Suppose that we use double hashing to resolve collisions—that is, we use the hash function $h(k, i) = (h_1(k) + ih_2(k)) \bmod m$. Show that if m and $h_2(k)$ have greatest common divisor $d \geq 1$ for some key k , then an unsuccessful search for key k examines $(1/d)$ th of the hash table before returning to slot $h_1(k)$. Thus, when $d = 1$, so that m and $h_2(k)$ are relatively prime, the search may examine the entire hash table.

Suppose $d = \gcd(m, h_2(k))$, the LCM $l = m \cdot h_2(k)/d$.

Since $d|h_2(k)$, then $m \cdot h_2(k)/d \bmod m = 0 \cdot (h_2(k)/d \bmod m) = 0$, therefore $(l + ih_2(k)) \bmod m = ih_2(k) \bmod m$, which means $ih_2(k) \bmod m$ has a period of m/d .

11.4-5 *

Consider an open-address hash table with a load factor α . Find the nonzero value α for which the expected number of probes in an unsuccessful search equals twice the expected number of probes in a successful search. Use the upper bounds given by Theorems 11.6 and 11.8 for these expected numbers of probes.

$$\frac{1}{1 - \alpha} = 2 \cdot \frac{1}{\alpha} \ln \frac{1}{1 - \alpha}$$

$$\alpha = 0.71533$$

11.5 Perfect hashing

11.5-1 *

Suppose that we insert n keys into a hash table of size m using open addressing and uniform hashing. Let $p(n, m)$ be the probability that no collisions occur. Show that $p(n, m) \leq e^{-n(n-1)/2m}$.

$$p(n, m) = \frac{m}{m} \cdot \frac{m-1}{m} \cdots \frac{m-n+1}{m} = \frac{m \cdot (m-1) \cdots (m-n+1)}{m^n}$$

$$\begin{aligned} (m-i) \cdot (m-n+i) &= \left(m - \frac{n}{2} + \frac{n}{2} - i\right) \cdot \left(m - \frac{n}{2} - \frac{n}{2} + i\right) \\ &= \left(m - \frac{n}{2}\right)^2 - \left(i - \frac{n}{2}\right)^2 \\ &\leq \left(m - \frac{n}{2}\right)^2 \end{aligned}$$

$$\begin{aligned} p(n, m) &\leq \frac{m \cdot \left(m - \frac{n}{2}\right)^{n-1}}{m^n} \\ &= \left(1 - \frac{n}{2m}\right)^{n-1} \end{aligned}$$

Based on equation (3.12), $e^x \geq 1 + x$,

$$\begin{aligned} p(n, m) &\leq \left(e^{-n/2m}\right)^{n-1} \\ &= e^{-n(n-1)/2m} \end{aligned}$$

Problems

11-1 Longest-probe bound for hashing

Suppose that we use an open-addressed hash table of size m to store $n \leq m/2$ items.

- a. Assuming uniform hashing, show that for $i = 1, 2, \dots, n$, the probability is at most 2^{-k} that the i th insertion requires strictly more than k probes.

$$\begin{aligned}\Pr\{X_i > k\} &= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-k+1}{m-k+1} \\ &\leq \left(\frac{n}{m}\right)^k \\ &\leq \left(\frac{1}{2}\right)^k \\ &= 2^{-k}\end{aligned}$$

- b. Show that for $i = 1, 2, \dots, n$, the probability is $O(1/n^2)$ that the i th insertion requires more than $2 \lg n$ probes.

$$\Pr\{X_i > 2 \lg n\} \leq 2^{-2 \lg n} = 1/n^2 = O(1/n^2)$$

Let the random variable X_i denote the number of probes required by the i th insertion.

You have shown in part (b) that $\Pr\{X_i > 2 \lg n\} = O(1/n^2)$. Let the random variable $X = \max_{1 \leq i \leq n} X_i$ denote the maximum number of probes required by any of the n insertions.

- c. Show that $\Pr\{X > 2 \lg n\} = O(1/n)$.

$$\Pr\{X > 2 \lg n\} \leq \sum_{i=1}^n 1/n^2 = 1/n = O(1/n)$$

- d. Show that the expected length $E[X]$ of the longest probe sequence is $O(\lg n)$.

$$\begin{aligned}
\mathbb{E}[X] &= \sum_{i=1}^n i \cdot \Pr\{X = i\} \\
&= \sum_{i=1}^{2 \lg n} i \cdot \Pr\{X = i\} + \sum_{i=2 \lg n+1}^n i \cdot \Pr\{X = i\} \\
&\leq 2 \lg n \cdot \sum_{i=1}^{2 \lg n} \Pr\{X = i\} + n \cdot \sum_{i=2 \lg n+1}^n \Pr\{X = i\} \\
&\leq 2 \lg n \cdot 1 + n \cdot 1/n \\
&= 2 \lg n + 1 \\
&= O(\lg n)
\end{aligned}$$

11-2 Slot-size bound for chaining

Suppose that we have a hash table with n slots, with collisions resolved by chaining, and suppose that n keys are inserted into the table. Each key is equally likely to be hashed to each slot. Let M be the maximum number of keys in any slot after all the keys have been inserted. Your mission is to prove an $O(\lg n / \lg \lg n)$ upper bound on $\mathbb{E}[M]$, the expected value of M .

a. Argue that the probability Q_k that exactly k keys hash to a particular slot is given by

$$Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k}$$

Obviously,

b. Let P_k be the probability that $M = k$, that is, the probability that the slot containing the most keys contains k keys. Show that $P_k \leq nQ_k$.

nQ_k is the probability that at least one slot contains k keys, thus $P_k \leq nQ_k$.

c. Use Stirling's approximation, equation (3.18), to show that $Q_k < e^k/k^k$.

$$\begin{aligned}
Q_k &= Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k} \\
&= \frac{(n-1)^{n-k}}{n^n} \cdot \frac{n \cdot (n-1) \cdots (n-k+1)}{k!} \\
&\leq \frac{(n-1)^{n-k}}{n^n} \cdot \frac{n^k}{\sqrt{2\pi k} \left(\frac{k}{e}\right)^k} \\
&= \left(\frac{n-1}{n}\right)^{n-k} \cdot \frac{(e/k)^k}{\sqrt{2\pi k}} \\
&\leq e^k / k^k
\end{aligned}$$

d. Show that there exists a constant $c > 1$ such that $Q_{k_0} < 1/n^3$ for $k_0 = c \lg n / \lg \lg n$. Conclude that $P_k < 1/n^2$ for $k \geq k_0 = c \lg n / \lg \lg n$.

$$\lg Q_{k_0} = \frac{c \lg n (\lg e - \lg c)}{\lg \lg n} + c \lg n \cdot \left(\frac{\lg \lg \lg n}{\lg \lg n} - 1 \right)$$

$$\frac{\lg \lg \lg n}{\lg \lg n} \text{ is } \frac{1}{e \log(2)} \approx 0.5307$$

The maximum of $\frac{\lg \lg \lg n}{\lg \lg n}$ is $e \log(2)$, and converge to 0 when $n \rightarrow \infty$. For a large n , if $c > 3$, the first term is negative and

$$\lg Q_{k_0} < -3 \lg n = \lg \frac{1}{n^3}$$

$$\therefore Q_{k_0} < \frac{1}{n^3}$$

$$\therefore P_k \leq n Q_k < \frac{1}{n^2}$$

e. Argue that

$$E[M] \leq \Pr \left\{ M > \frac{c \lg n}{\lg \lg n} \right\} \cdot n + \Pr \left\{ M \leq \frac{c \lg n}{\lg \lg n} \right\} \cdot \frac{c \lg n}{\lg \lg n}$$

Conclude that $E[M] = O(\lg n / \lg \lg n)$.

$$\begin{aligned}
\mathbb{E}[M] &= \sum_{i=1}^n \Pr\{M = i\} \cdot i \\
&= \frac{c \lg n}{\lg \lg n} \sum_{i=1}^n \Pr\{M = i\} \cdot i + \sum_{i=\frac{c \lg n}{\lg \lg n} + 1}^n \Pr\{M = i\} \cdot i \\
&\leq \frac{c \lg n}{\lg \lg n} \sum_{i=1}^n \Pr\{M = i\} \cdot \frac{c \lg n}{\lg \lg n} + \sum_{i=\frac{c \lg n}{\lg \lg n} + 1}^n \Pr\{M = i\} \cdot n \\
&= \Pr\left\{M \leq \frac{c \lg n}{\lg \lg n}\right\} \cdot \frac{c \lg n}{\lg \lg n} + \Pr\left\{M > \frac{c \lg n}{\lg \lg n}\right\} \cdot n \\
&< 1 \cdot \frac{c \lg n}{\lg \lg n} + \frac{1}{n^2} \cdot n \\
&= \frac{c \lg n}{\lg \lg n} + \frac{1}{n} \\
&= O(\lg n / \lg \lg n)
\end{aligned}$$

$\left(\lim_{n \rightarrow \infty} \frac{\frac{1}{n}}{\frac{\lg n}{\lg \lg n}} = \lim_{n \rightarrow \infty} \frac{\lg(\lg n)}{\lg(n^n)} = 0 \right)$

11-3 Quadratic probing

Suppose that we are given a key k to search for in a hash table with positions $0, 1, \dots, m - 1$, and suppose that we have a hash function h mapping the key space into the set $\{0, 1, \dots, m - 1\}$. The search scheme is as follows:

1. Compute the value $j = h(k)$, and set $i = 0$.
2. Probe in position j for the desired key k . If you find it, or if this position is empty, terminate the search.
3. Set $i = i + 1$. If i now equals m , the table is full, so terminate the search.
Otherwise, set $j = (i + j) \bmod m$, and return to step 2.

Assume that m is a power of 2.

- a.** Show that this scheme is an instance of the general "quadratic probing" scheme by exhibiting the appropriate constants c_1 and c_2 for equation (11.5).

The i th probing is equivalent to $(j + \frac{i(i+1)}{2}) \bmod m$, thus $c_1 = 1/2$, $c_2 = 1/2$.

- b.** Prove that this algorithm examines every table position in the worst case.

Suppose there are two probing i and j , and $0 \leq j < i < m$.

Suppose the two probing examine the same position, then:

$$\begin{aligned}(i + i^2) - (j + j^2) &= 2km \quad (k \geq 0) \\ (i + j + 1)(i - j) &= 2km\end{aligned}$$

Since $i > j$, then $k \neq 0$.

Note that m is a power of 2.

If i and j are both even or both odd, then only $(i - j)$ could be even, since $i < m$, $(i - j) < m < 2m$, thus $2m$ could not be a factor of $(i - j)$.

If one of i and j is odd and the other is even, then only $(i + j + 1)$ could be even, since $i < m$, $(i + j + 1) < 2m$, thus $2m$ could not be a factor of $(i + j + 1)$.

Therefore i and j could not probe the same position, this algorithm examines every table position in the worst case.

11-4 Hashing and authentication

Let \mathcal{H} be a class of hash functions in which each hash function $h \in \mathcal{H}$ maps the universe U of keys to $\{0, 1, \dots, m - 1\}$. We say that \mathcal{H} is **k -universal** if, for every fixed sequence of k distinct keys $\langle x^{(1)}, x^{(2)}, \dots, x^{(k)} \rangle$ and for any h chosen at random from \mathcal{H} , the sequence $\langle h(x^{(1)}), h(x^{(2)}), \dots, h(x^{(k)}) \rangle$ is equally likely to be any of the m^k sequences of length k with elements drawn from $\{0, 1, \dots, m - 1\}$.

a. Show that if the family \mathcal{H} of hash functions is 2-universal, then it is universal.

The number of hash functions for which $h(k) = h(l)$ is $\frac{m}{m^2} |\mathcal{H}| = \frac{1}{m} |\mathcal{H}|$, therefore the family is universal.

b. Suppose that the universe U is the set of n -tuples of values drawn from $\mathbb{Z}_p = \{0, 1, \dots, p - 1\}$, where p is prime. Consider an element $x = \langle x_0, x_1, \dots, x_{n-1} \rangle \in U$. For any n -tuple $a = \langle a_0, a_1, \dots, a_{n-1} \rangle \in U$, define the hash function h_a by

$$h_a(x) = \left(\sum_{j=0}^{n-1} a_j x_j \right) \bmod p$$

Let $\mathcal{H} = \{h_a\}$. Show that \mathcal{H} is universal, but not 2-universal.

For $x = \langle 0, 0, \dots, 0 \rangle$, \mathcal{H} could not be 2-universal.

c. Suppose that we modify \mathcal{H} slightly from part (b): for any $a \in U$ and for any $b \in \mathbb{Z}_p$, define

$$h'_{ab}(x) = \left(\sum_{j=0}^{n-1} a_j x_j \right) \bmod p$$

and $\mathcal{H}' = \{h'_{ab}\}$. Argue that \mathcal{H}' is 2-universal.

d. Suppose that Alice and Bob secretly agree on a hash function h from 2-universal family \mathcal{H} of hash functions. Each $h \in \mathcal{H}$ maps from a universe of keys U to \mathbb{Z}_p , where p is a prime. Later, Alice sends a message m to Bob over the Internet, where $m \in U$. She authenticates this message to Bob by also sending an authentication tag $t = h(m)$, and Bob checks that the pair (m, t) he receives indeed satisfies $t = h(m)$. Suppose that an adversary intercepts (m, t) en route and tries to fool Bob by replacing the pair (m, t) with a different pair (m', t') . Argue that the probability that the adversary succeeds in fooling Bob into accepting (m', t') is at most $1/p$, no matter how much computing power the adversary has, and even if the adversary knows the family \mathcal{H} of hash functions used.

Since \mathcal{H} is 2-universal, every pair of $\langle t, t' \rangle$ is equally likely to appear, thus t' could be any value from \mathbb{Z}_p . Even the adversary knows \mathcal{H} , since \mathcal{H} is 2-universal, then \mathcal{H} is universal, the probability of choosing a hash function that $h(k) = h(l)$ is at most $1/p$, therefore the probability is at most $1/p$.

12 Binary Search Trees

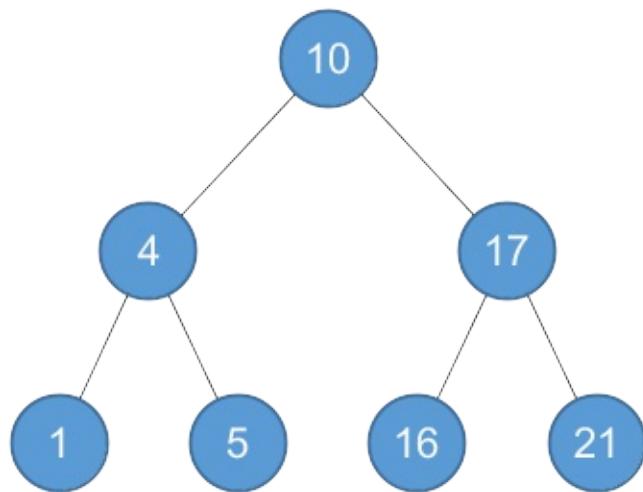
- 12.1 What is a binary search tree?
- 12.2 Querying a binary search tree
- 12.3 Insertion and deletion
- 12.4 Randomly built binary search trees
- Problems

12.1 What is a binary search tree?

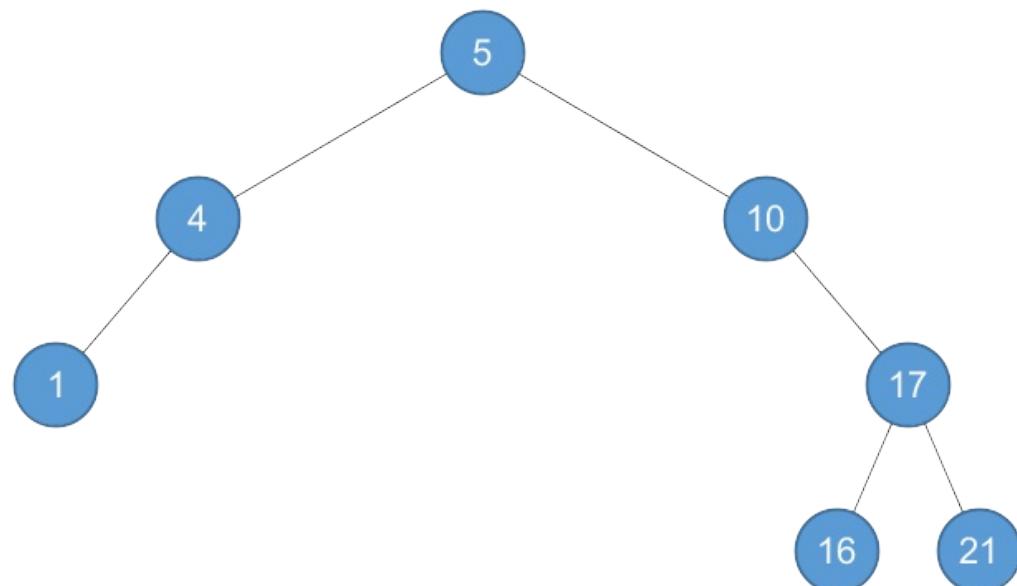
12.1-1

For the set of $\langle 1, 4, 5, 10, 16, 17, 21 \rangle$ of keys, draw binary search trees of heights 2, 3, 4, 5, and 6.

<https://github.com/CyberZHCG>

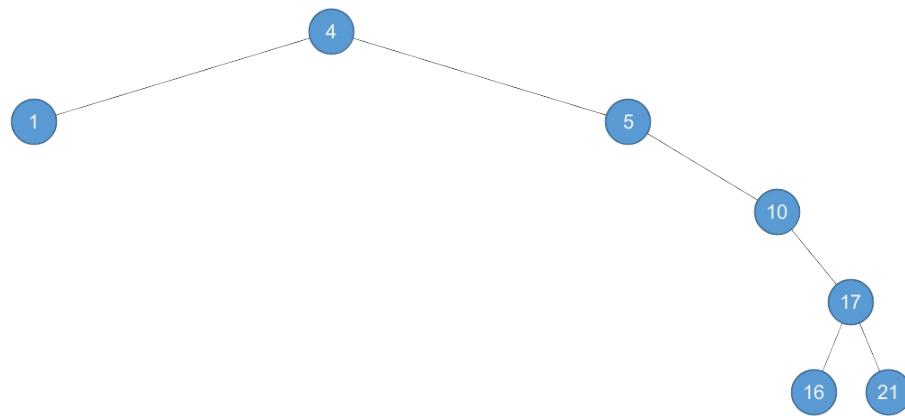


<https://github.com/CyberZHCG>

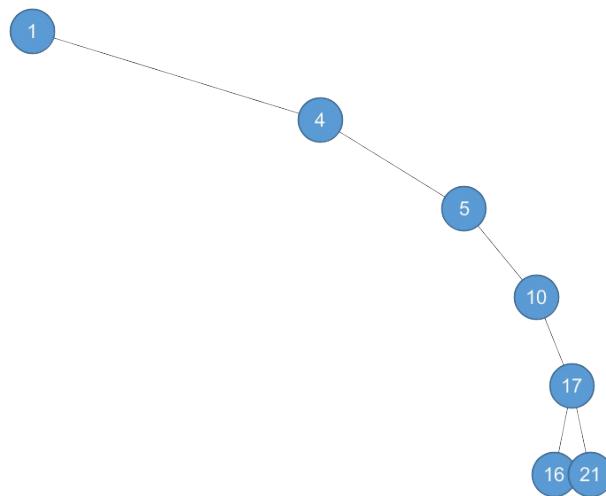


12.1 What is a binary search tree?

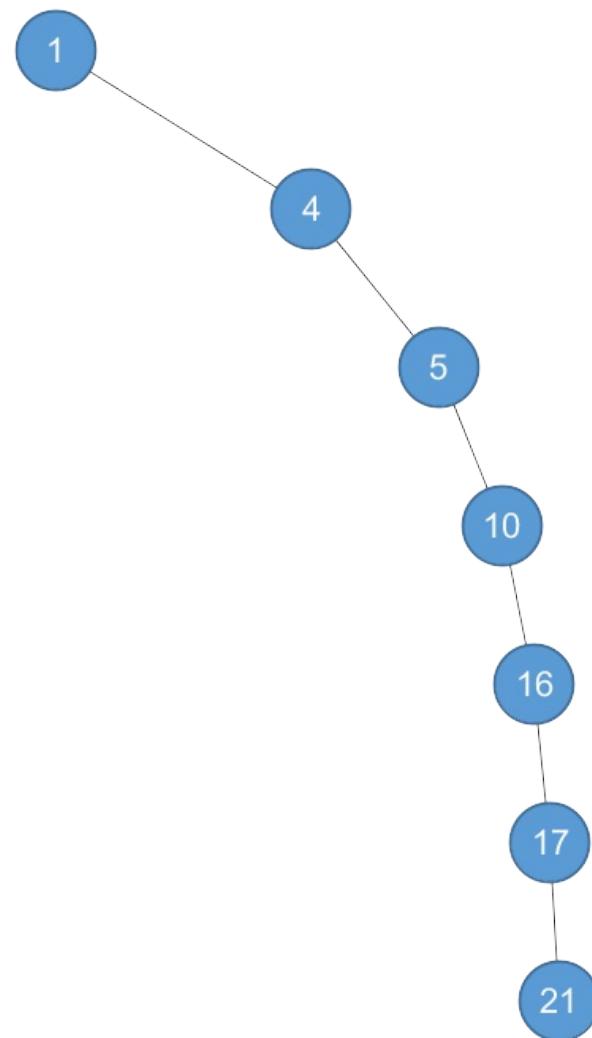
https://algs4.csail.mit.edu/2016.1



https://algs4.csail.mit.edu/2016.1



<https://github.com/CyberZH/CLRS-PDF-Notes>



12.1-2

What is the difference between the binary-search-tree property and the min-heap property (see page 153)? Can the min-heap property be used to print out the keys of an n -node tree in sorted order in $O(n)$ time? Show how, or explain why not.

No, heap needs $O(n \lg n)$ time.

12.1-3

Give a nonrecursive algorithm that performs an inorder tree walk.

12.1 What is a binary search tree?

```
class TreeNode:  
    def __init__(self, val, left=None, right=None):  
        self.val = val  
        self.left = left  
        self.right = right  
  
def inorder_tree_walk(root):  
    stack = []  
    while len(stack) > 0 or root is not None:  
        if root is None:  
            root = stack[-1]  
            del stack[-1]  
            print(root.val)  
            root = root.right  
        else:  
            stack.append(root)  
            root = root.left
```

12.1-4

Give recursive algorithms that perform preorder and postorder tree walks in $\Theta(n)$ time on a tree of n nodes.

```
class TreeNode:  
    def __init__(self, val, left=None, right=None):  
        self.val = val  
        self.left = left  
        self.right = right  
  
def preorder_tree_walk(root):  
    if root is not None:  
        print(root.val)  
        preorder_tree_walk(root.left)  
        preorder_tree_walk(root.right)  
  
def postorder_tree_walk(root):  
    if root is not None:  
        postorder_tree_walk(root.left)  
        postorder_tree_walk(root.right)  
        print(root.val)
```

12.1-5

Argue that since sorting n elements takes $\Omega(n \lg n)$ time in the worst case in the comparison model, any comparison-based algorithm for constructing a binary search tree from an arbitrary list of n elements takes $\Omega(n \lg n)$ time in the worst case.

If we construct the binary search tree by comparison-based algorithm using less than $\Omega(n \lg n)$ time, since the inorder tree walk is $\Theta(n)$, then we can get the sorted elements in less than $\Omega(n \lg n)$ time, which contradicts the fact that sorting n elements takes $\Omega(n \lg n)$ time in the worst case.

12.2 Querying a binary search tree

12.2-1

Suppose that we have numbers between 1 and 1000 in a binary search tree, and we want to search for the number 363. Which of the following sequences could not be the sequence of nodes examined?

- a. 2, 252, 401, 398, 330, 344, 397, 363.
- b. 924, 220, 911, 244, 898, 258, 362, 363.
- c. 925, 202, 911, 240, 912, 245, 363.
- d. 2, 399, 387, 219, 266, 382, 381, 278, 363.
- e. 935, 278, 347, 621, 299, 392, 358, 363.

- c is impossible since $911 < 912$.
- e is impossible since $299 < 347$.

12.2-2

Write recursive versions of TREE-MINIMUM and TREE-MAXIMUM.

```
class TreeNode:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def tree_minimum(root):
    if root is None:
        return None
    if root.left is None:
        return root.val
    return tree_minimum(root.left)

def tree_maximum(root):
    if root is None:
        return None
    if root.right is None:
        return root.val
    return tree_maximum(root.right)
```

12.2-3

Write the TREE-PREDECESSOR procedure.

```

class TreeNode:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.parent = None
        self.left = left
        self.right = right
        if left is not None:
            left.parent = self
        if right is not None:
            right.parent = self

def tree_maximum(root):
    if root is None:
        return None
    if root.right is None:
        return root
    return tree_maximum(root.right)

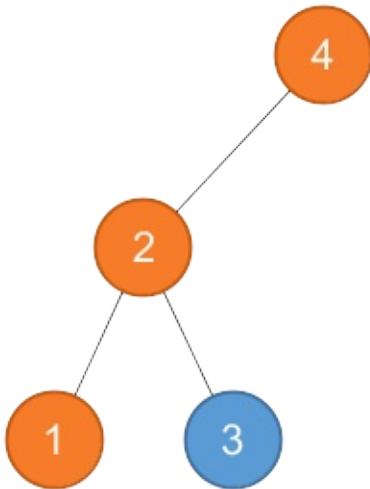
def tree_predecessor(root):
    if root is None:
        return None
    if root.left is not None:
        return tree_maximum(root.left)
    p = root.parent
    while p is not None and root == p.left:
        root = p
        p = p.parent
    return p

```

12.2-4

Professor Bunyan thinks he has discovered a remarkable property of binary search trees. Suppose that the search for key k in a binary search tree ends up in a leaf. Consider three sets: A , the keys to the left of the search path; B , the keys on the search path; and C , the keys to the right of the search path. Professor Bunyan claims that any three keys $a \in A$, $b \in B$, and $c \in C$ must satisfy $a \leq b \leq c$. Give a smallest possible counterexample to the professor's claim.

<https://github.com/CyberZH/CS-Notes/blob/main/12-Binary%20Search%20Tree/12.2%20Querying%20a%20binary%20search%20tree.md>



$3 < 4$

12.2-5

Show that if a node in a binary search tree has two children, then its successor has no left child and its predecessor has no right child.

If its successor has left child, then the left child is less than successor and it's larger than the node, thus the successor is not the successor.

12.2-6

Consider a binary search tree T whose keys are distinct. Show that if the right subtree of a node x in T is empty and x has a successor y , then y is the lowest ancestor of x whose left child is also an ancestor of x . (Recall that every node is its own ancestor.)

TREE-SUCCESSOR

12.2-7

An alternative method of performing an inorder tree walk of an n -node binary search tree finds the minimum element in the tree by calling TREE-MINIMUM and then making $n - 1$ calls to TREE-SUCCESSOR. Prove that this algorithm runs in $\Theta(n)$ time.

Based on 12.2-8, it takes $O(h + n + n - 1)$ time, therefore it's $\Theta(n)$.

12.2-8

Prove that no matter what node we start at in a height- h binary search tree, k successive calls to TREE-SUCCESSOR take $O(k + h)$ time.

Suppose x is the starting node and y is the ending node. The distance between x and y is at most $2h$, and all the edges connecting the k nodes are visited twice, therefore it takes $O(k + h)$ time.

12.2-9

Let T be a binary search tree whose keys are distinct, let x be a leaf node, and let y be its parent. Show that $y.key$ is either the smallest key in T larger than $x.key$ or the largest key in T smaller than $x.key$.

TREE-SUCCESSOR

12.3 Insertion and deletion

12.3-1

Give a recursive version of the TREE-INSERT procedure.

```
class TreeNode:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.parent = None
        self.left = left
        self.right = right
        if left is not None:
            left.parent = self
        if right is not None:
            right.parent = self

def insert(root, x):
    if root is None:
        return TreeNode(x)
    if root.val > x:
        root.left = insert(root.left, x)
        root.left.parent = root
    elif root.val < x:
        root.right = insert(root.right, x)
        root.right.parent = root
    return root
```

12.3-2

Suppose that we construct a binary search tree by repeatedly inserting distinct values into the tree. Argue that the number of nodes examined in searching for a value in the tree is one plus the number of nodes examined when the value was first inserted into the tree.

Obviously

12.3-3

We can sort a given set of n numbers by first building a binary search tree containing these numbers (using TREE-INSERT repeatedly to insert the numbers one by one) and then printing the numbers by an inorder tree walk. What are the worstcase and best-case running times for this sorting algorithm?

Worst: chain, $O(n^2)$

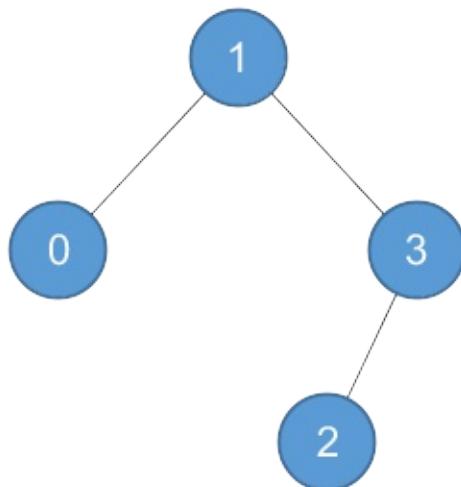
Best: $\Theta(n \lg n)$

12.3-4

Is the operation of deletion "commutative" in the sense that deleting x and then y from a binary search tree leaves the same tree as deleting y and then x ? Argue why it is or give a counterexample.

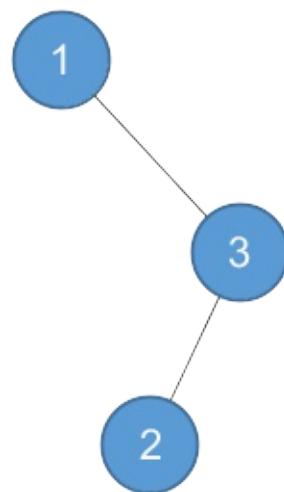
No.

<https://github.com/CyberZH/CLRS-Python-Solutions/blob/main/problems/12.3-4.py>

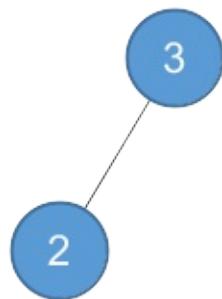


Delete 0 then delete 1:

<https://github.com/CyberZH3>



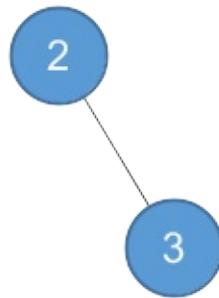
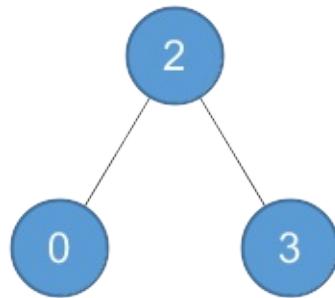
<https://github.com/CyberZH3>



Delete 1 then delete 0:

<https://github.com/CyberZH3>

<https://github.com/CyberZH3>



12.3-5

Suppose that instead of each node x keeping the attribute $x.p$, pointing to x 's parent, it keeps $x.succ$, pointing to x 's successor. Give pseudocode for SEARCH, INSERT, and DELETE on a binary search tree T using this representation. These procedures should operate in time $O(h)$, where h is the height of the tree T .

In SEARCH and INSERT, we do not need to know the parent of x .

```
def get_parent(root, node):
    if node is None:
        return None
    a = tree_successor(tree_maximum(node))
    if a is None:
        a = root
    else:
        if a.left == node:
            return a
        a = a.left
    while a is not None and a.right != node:
        a = a.right
    return a
```

Therefore we can find x 's parent in $O(h)$, DELETE is $O(h + h) = O(h)$.

12.3-6

When node z in TREE-DELETE has two children, we could choose node y as its predecessor rather than its successor. What other changes to TREE-DELETE would be necessary if we did so? Some have argued that a fair strategy, giving equal priority to predecessor and successor, yields better empirical performance. How might TREE-DELETE be changed to implement such a fair strategy?

Randomly choose predecessor and successor.

12.4 Randomly built binary search trees

12.4-1

Prove equation (12.3).

$$\sum_{i=0}^{n-1} \binom{i+3}{3} = \binom{n+3}{4}$$

$$\begin{aligned} \sum_{i=0}^{n-1} \binom{i+3}{3} &= \sum_{i=0}^{n-1} \frac{(i+3)(i+2)(i+1)}{6} \\ &= \frac{1}{6} \sum_{i=0}^{n-1} i^3 + 6i^2 + 11i + 6 \\ &= \frac{1}{6} \left(\frac{(n-1)^2 n^2}{4} + \frac{6(n-1)n(2n-1)}{6} + \frac{11n(n-1)}{2} + 6n \right) \\ &= \frac{n(n+1)(n+2)(n+3)}{24} \\ &= \binom{n+3}{4} \end{aligned}$$

12.4-2

Describe a binary search tree on n nodes such that the average depth of a node in the tree is $\Theta(\lg n)$ but the height of the tree is $\omega(\lg n)$. Give an asymptotic upper bound on the height of an n -node binary search tree in which the average depth of a node is $\Theta(\lg n)$.

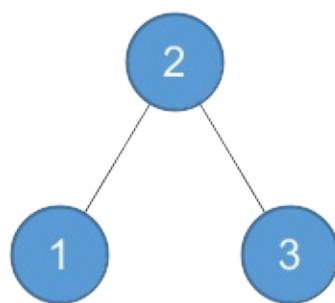
$$\Theta(\sqrt{n \lg n})$$

12.4-3

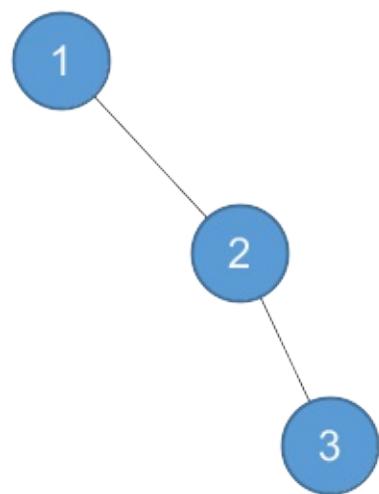
Show that the notion of a randomly chosen binary search tree on n keys, where each binary search tree of n keys is equally likely to be chosen, is different from the notion of a randomly built binary search tree given in this section.

For $n = 3$, there are 5 binary search trees. However, if we build the trees will a random permutation, the first tree will built twice.

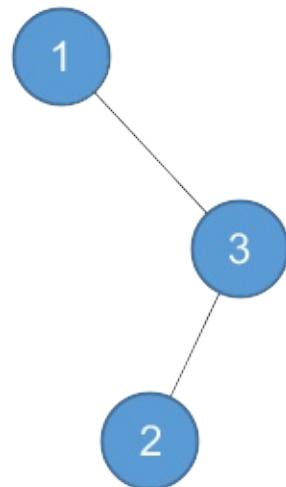
<https://github.com/CyberZH/>



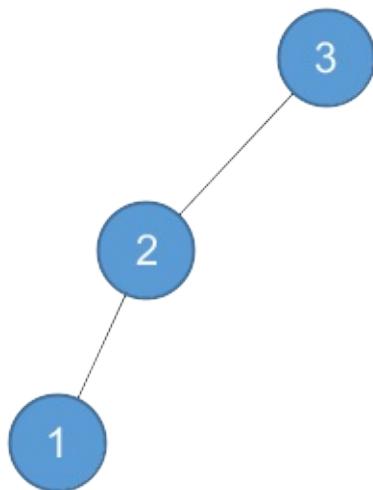
<https://github.com/CyberZH/>



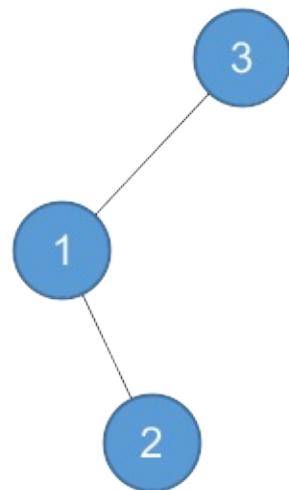
<https://github.com/CyberZH/>



<https://github.com/CyberZH/>



<https://github.com/CyberZH/>



12.4-4

Show that the function $f(x) = 2^x$ is convex.

$$\begin{aligned} f(x) + f(y) - 2f\left(\frac{x+y}{2}\right) &= 2^x + 2^y - 2 \cdot 2^{(x+y)/2} \\ &= (2^{x/2})^2 + (2^{y/2})^2 - 2 \cdot 2^{x/2} \cdot 2^{y/2} \\ &= (2^{x/2} - 2^{y/2})^2 \geq 0 \end{aligned}$$

Therefore $f(x) = 2^x$ is convex.

12.4-5 *

Consider RANDOMIZED-QUICKSORT operating on a sequence of n distinct input numbers. Prove that for any constant $k > 0$, all but $O(1/n^k)$ of the $n!$ input permutations yield an $O(n \lg n)$ running time.

...

Problems

12-1 Binary search trees with equal keys

Equal keys pose a problem for the implementation of binary search trees.

- a. What is the asymptotic performance of TREE-INSERT when used to insert n items with identical keys into an initially empty binary search tree?

$\Theta(n)$

- b. Keep a boolean flag $x.b$ at node x , and set x to either $x.left$ or $x.right$ based on the value of $x.b$, which alternates between FALSE and TRUE each time we visit x while inserting a node with the same key as x .

$\Theta(n \lg n)$

- c. Keep a list of nodes with equal keys at x , and insert z into the list.

If linked list is used, it could be $\Theta(1)$.

- d. Randomly set x to either $x.left$ or $x.right$. (Give the worst-case performance and informally derive the expected running time.)

Worst: $\Theta(n)$

Expected: $\Theta(n \lg n)$

12-2 Radix trees

Given two strings $a = a_0 a_1 \dots a_p$ and $b = b_0 b_1 \dots b_q$, where each a_i and each b_j is in some ordered set of characters, we say that string a is lexicographically less than string b if either

1. there exists an integer j , where $0 \leq j \leq \min(p, q)$, such that $a_i = b_i$ for all $i = 0, 1, \dots, j - 1$ and $a_j < b_j$, or
2. $p < q$ and $a_i = b_i$ for all $i = 0, 1, \dots, p$.

The radix tree data structure shown in Figure 12.5 stores the bit strings 1011, 10, 011, 100, and 0. When searching for a key $a = a_0 a_1 \dots a_p$, we go left at a node of depth i if $a_i = 0$ and right if $a_i = 1$. Let S be a set of distinct bit strings whose lengths sum to n . Show how to use a radix tree to sort S lexicographically in $\Theta(n)$ time. For the example in Figure 12.5, the output of the sort should be the sequence 0, 011, 10, 100, 1011.

Insert all the bit strings into radix tree costs $\Theta(n)$, then use preorder tree walk to sort the strings costs $\Theta(n)$, the total cost is still $\Theta(n)$.

```

class TreeNode:
    def __init__(self, val, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


class RadixTree:
    def __init__(self):
        self.root = None

    def insert(self, a):
        self.root = self.insert_rec(self.root, a, 0)

    def insert_rec(self, root, a, idx):
        if idx == len(a):
            if root is None:
                return TreeNode(a)
            root.val = a
            return root
        if root is None:
            root = TreeNode(None)
        if a[idx] == '0':
            root.left = self.insert_rec(root.left, a, idx+1)
        else:
            root.right = self.insert_rec(root.right, a, idx+1)
        return root

    def sorted(self):
        self.sorted_list = []
        self.sorted_rec(self.root)
        return self.sorted_list

    def sorted_rec(self, root):
        if root is None:
            return
        if root.val is not None:
            self.sorted_list.append(root.val)
        self.sorted_rec(root.left)
        self.sorted_rec(root.right)

def sort_strings(strs):
    radix_tree = RadixTree()
    for s in strs:
        radix_tree.insert(s)
    return radix_tree.sorted()

```

12-3 Average node depth in a randomly built binary search tree

In this problem, we prove that the average depth of a node in a randomly built binary search tree with n nodes is $O(\lg n)$. Although this result is weaker than that of Theorem 12.4, the technique we shall use reveals a surprising similarity between the building of a binary search tree and the execution of RANDOMIZED-QUICKSORT from Section 7.3.

We define the **total path length** $P(T)$ of a binary tree T as the sum, over all nodes x in T , of the depth of node x , which we denote by $d(x, T)$.

a. Argue that the average depth of a node in T is

$$\frac{1}{n} \sum_{x \in T} d(x, T) = \frac{1}{n} P(T)$$

Obviously.

Thus, we wish to show that the expected value of $P(T)$ is $O(n \lg n)$.

b. Let T_L and T_R denote the left and right subtrees of tree T , respectively. Argue that if T has n nodes, then

$$P(T) = P(T_L) + P(T_R) + n - 1$$

There are $n - 1$ nodes in $P(T_L)$ and $P(T_R)$, each increase by 1.

c. Let $P(n)$ denote the average total path length of a randomly built binary search tree with n nodes. Show that

$$P(n) = \frac{1}{n} \sum_{i=0}^{n-1} (P(i) + P(n-i-1) + n - 1)$$

The root is equally likely to be the rank in $[1, n]$.

d. Show how to rewrite $P(n)$ as

$$P(n) = \frac{2}{n} \sum_{k=1}^{n-1} P(k) + \Theta(n)$$

Each item $P(1), P(2), \dots, P(n)$ appears twice in the summation, and

$$\frac{1}{n} \sum_{i=0}^{n-1} n - 1 = \frac{(n-3)n}{2n} = \Theta(n)$$

- e.** Recalling the alternative analysis of the randomized version of quicksort given in Problem 7-3, conclude that $P(n) = O(n \lg n)$.

Based on Problem 7-3, $P(n) = O(n \lg n)$.

- f.** Describe an implementation of quicksort in which the comparisons to sort a set of elements are exactly the same as the comparisons to insert the elements into a binary search tree.

Choose the pivot that it has the lowest index in the original list.

12-4 Number of different binary trees

Let b_n denote the number of different binary trees with n nodes. In this problem, you will find a formula for b_n , as well as an asymptotic estimate.

- a.** Show that $b_0 = 1$ and that, for $n \geq 1$,

$$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k}$$

A root with two subtree.

- b.** Referring to Problem 4-4 for the definition of a generating function, let $B(x)$ be the generating function

$$B(x) = \sum_{n=0}^{\infty} b_n x^n$$

$$\begin{aligned} B(x)^2 &= (b_0 x^0 + b_1 x^1 + b_2 x^2 + \dots)^2 \\ &= b_0^2 x^0 + (b_0 b_1 + b_1 b_0) x^1 + (b_0 b_2 + b_1 b_1 + b_2 b_0) x^2 + \dots \\ &= \sum_{k=0}^0 b_k b_{0-k} x^0 + \sum_{k=0}^1 b_k b_{1-k} x^1 + \sum_{k=0}^2 b_k b_{2-k} x^2 + \dots \end{aligned}$$

$$\begin{aligned}
xB(x)^2 + 1 &= 1 + \sum_{k=0}^0 b_k b_{1-1-k} x^1 + \sum_{k=0}^2 b_k b_{2-1-k} x^3 + \sum_{k=0}^2 b_k b_{3-1-k} x^2 + \dots \\
&= 1 + b_1 x^1 + b_2 x^2 + b_3 x^3 + \dots \\
&= b_0 x^0 + b_1 x^1 + b_2 x^2 + b_3 x^3 + \dots \\
&= \sum_{n=0}^{\infty} b_n x^n \\
&= B(x)
\end{aligned}$$

Show that $B(x) = xB(x)^2 + 1$, and hence one way to express $B(x)$ in closed form is

$$B(x) = \frac{1}{2x}(1 - \sqrt{1 - 4x})$$

$$\begin{aligned}
xB(x)^2 + 1 &= x \cdot \frac{1}{4x^2} (1 + 1 - 4x - 2\sqrt{1 - 4x}) + 1 \\
&= \frac{1}{4x} (2 - 2\sqrt{1 - 4x}) - 1 + 1 \\
&= \frac{1}{2x} (1 - \sqrt{1 - 4x}) \\
&= B(x)
\end{aligned}$$

The **Taylor expansion** of $f(x)$ around the point $x = a$ is given by

$$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!} (x - a)^k$$

where $f^{(k)}(a)$ is the k th derivative of f evaluated at x .

c. Show that

$$b_n = \frac{1}{n+1} \binom{2n}{n}$$

(the n th Catalan number) by using the Taylor expansion of $\sqrt{1 - 4x}$ around $x = 0$.

Let $f(x) = \sqrt{1 - 4x}$,

The numerator of the derivative is

$$\begin{aligned}
2 \cdot (1 \cdot 2) \cdot (3 \cdot 2) \cdot (5 \cdot 2) \cdots &= 2^k \cdot \prod_{i=0}^{k-2} (2k+1) \\
&= 2^k \cdot \frac{(2(k-1))!}{2^{k-1}(k-1)!} \\
&= \frac{2(2(k-1))!}{(k-1)!}
\end{aligned}$$

$$f(x) = 1 - 2x - 2x^2 - 4x^3 - 10x^4 - 28x^5 - \dots$$

$$\frac{2(2(k-1))!}{k!(k-1)!}$$

The coefficient is

$$\begin{aligned}
B(x) &= \frac{1}{2x}(1 - f(x)) \\
&= 1 + x + 2x^2 + 5x^3 + 14x^4 + \dots \\
&= \sum_{n=0}^{\infty} \frac{(2n)!}{(n+1)!n!} x \\
&= \sum_{n=0}^{\infty} \frac{1}{n+1} \frac{(2n)!}{n!n!} x \\
&= \sum_{n=0}^{\infty} \frac{1}{n+1} \binom{2n}{n} x
\end{aligned}$$

$$b_n = \frac{1}{n+1} \binom{2n}{n}$$

Therefore

d. Show that

$$b_n = \frac{4^n}{\sqrt{\pi} n^{3/2}} (1 + O(1/n))$$

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$$

Based on Stirling's approximation

$$\begin{aligned} b_n &= \frac{1}{n+1} \frac{(2n)!}{n!n!} \\ &\approx \frac{1}{n+1} \frac{\sqrt{4\pi n}(2n/e)^{2n}}{2\pi n(n/e)^{2n}} \\ &= \frac{1}{n+1} \frac{4^n}{\sqrt{\pi n}} \\ &= \left(\frac{1}{n} + \left(\frac{1}{n+1} - \frac{1}{n} \right) \right) \frac{4^n}{\sqrt{\pi n}} \\ &= \left(\frac{1}{n} - \frac{1}{n^2+n} \right) \frac{4^n}{\sqrt{\pi n}} \\ &= \frac{1}{n} \left(1 - \frac{1}{n+1} \right) \frac{4^n}{\sqrt{\pi n}} \\ &= \frac{4^n}{\sqrt{\pi} n^{3/2}} (1 + O(1/n)) \end{aligned}$$

13 Red-Black Trees

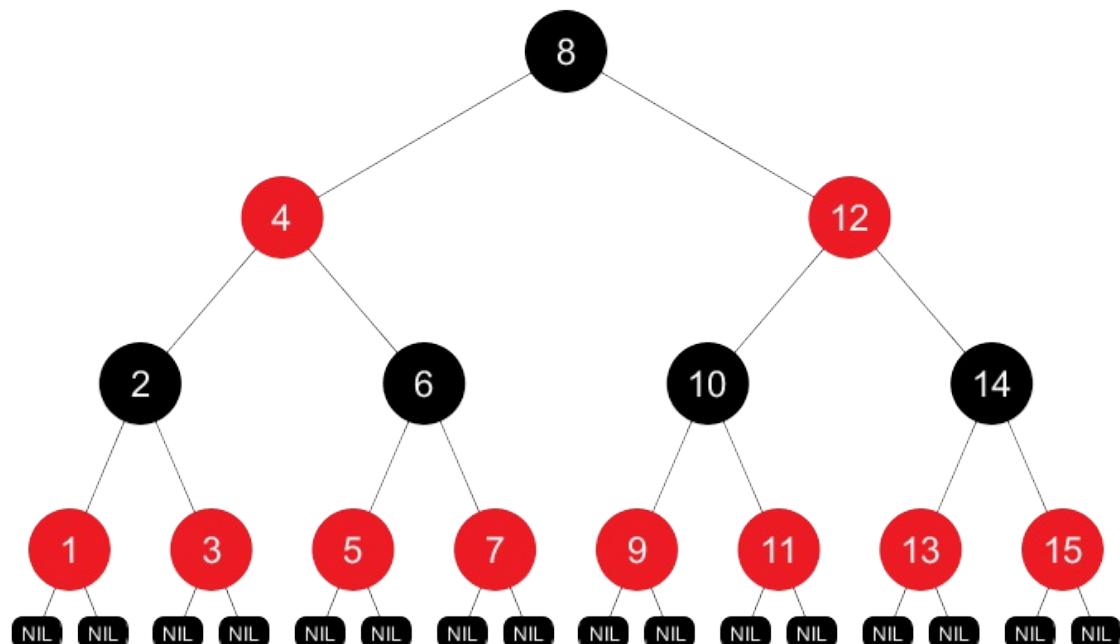
- 13.1 Properties of red-black trees
- 13.2 Rotations
- 13.3 Insertion
- 13.4 Deletion
- Problems

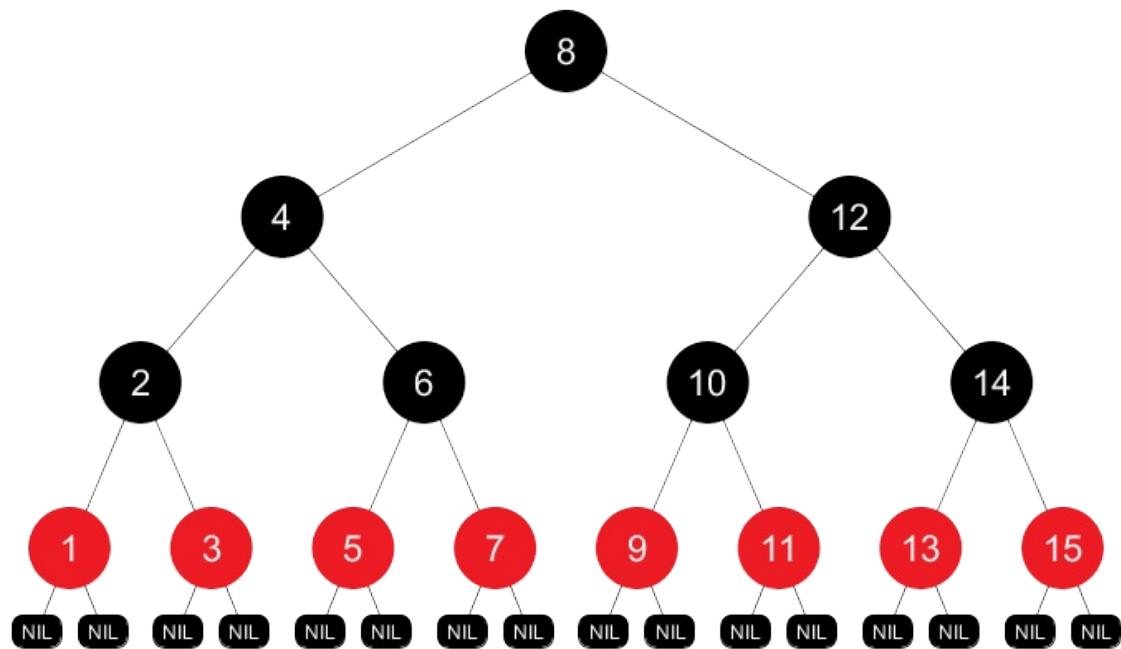
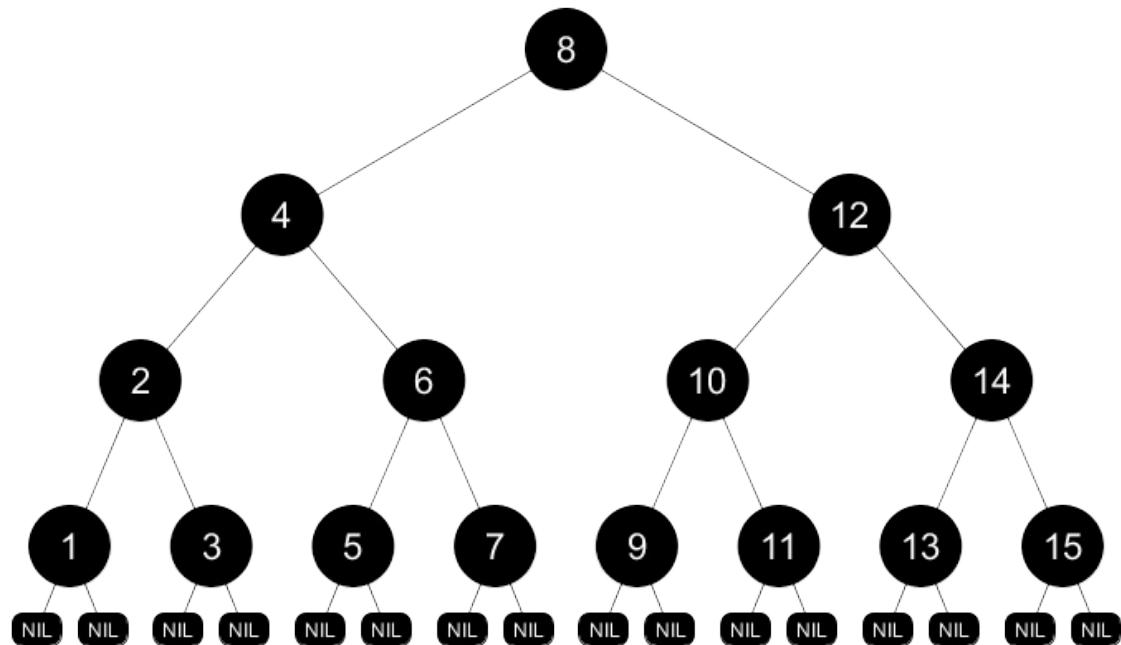
13.1 Properties of red-black trees

13.1-1

In the style of Figure 13.1(a), draw the complete binary search tree of height 3 on the keys $\{1, 2, \dots, 15\}$. Add the NIL leaves and color the nodes in three different ways such that the black-heights of the resulting red-black trees are 2, 3, and 4.

http://algs4.cs.princeton.edu/24rbt/

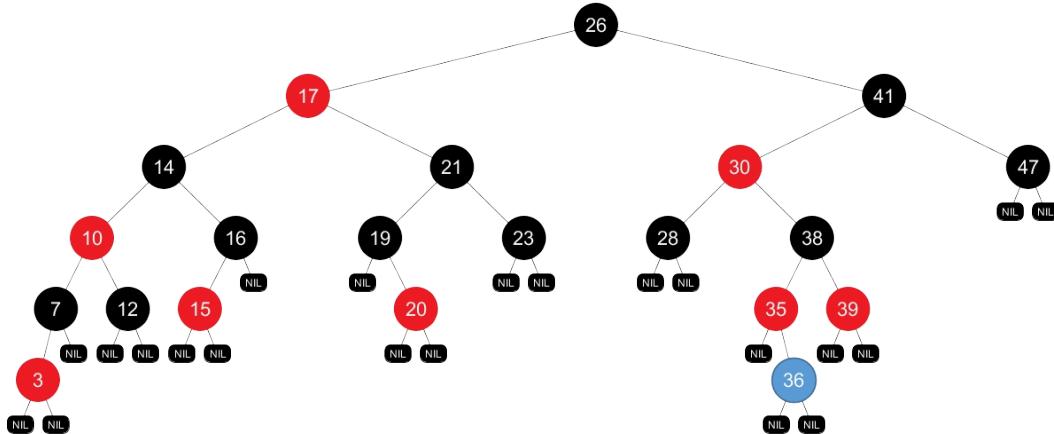


<https://github.com/CyberZH-CS/><https://github.com/CyberZH-CS/>

13.1-2

Draw the red-black tree that results after TREE-INSERT is called on the tree in Figure 13.1 with key 36. If the inserted node is colored red, is the resulting tree a red-black tree? What if it is colored black?

https://tinyurl.com/mwzqjw3y



If it is colored red, the tree doesn't satisfy property 4.

If it is colored black, the tree doesn't satisfy property 5.

13.1-3

Let us define a *relaxed red-black* tree as a binary search tree that satisfies red-black properties 1, 3, 4, and 5. In other words, the root may be either red or black. Consider a relaxed red-black tree T whose root is red. If we color the root of T black but make no other changes to T , is the resulting tree a red-black tree?

Obviously properties 1, 2, 3, 4 are satisfied.

Changing the root from red to black increases the number of black nodes in each path by 1, therefore they still have the same number of black nodes.

The resulting tree is a red-black tree.

13.1-4

Suppose that we "absorb" every red node in a red-black tree into its black parent, so that the children of the red node become children of the black parent. (Ignore what happens to the keys.) What are the possible degrees of a black node after all its red children are absorbed? What can you say about the depths of the leaves of the resulting tree?

The degree could be 2, 3, 4.

All the leaves have the same depth.

13.1-5

Show that the longest simple path from a node x in a red-black tree to a descendant leaf has length at most twice that of the shortest simple path from node x to a descendant leaf.

Since the paths contain the same number of black nodes $hb(x)$, we can insert one red node after each black nodes in a simple path and property 4 is satisfied, the resulting length is $2hb(x)$.

13.1-6

What is the largest possible number of internal nodes in a red-black tree with black-height k ? What is the smallest possible number?

The largest is the complete binary tree with height $2k$, which has $2^{2k} - 1$ internal nodes.

The smallest is a black chain with length k , which has k internal nodes.

13.1-7

Describe a red-black tree on n keys that realizes the largest possible ratio of red internal nodes to black internal nodes. What is this ratio? What tree has the smallest possible ratio, and what is the ratio?

The largest ratio is 2, each black node has two red children.

The smallest ratio is 0.

13.2 Rotations

13.2-1

Write pseudocode for RIGHT-ROTATE.

```

RIGHT-ROTATE(T, y)
1  x = y.left
2  y.left = x.right
3  if x.right != T.nil
4      x.right.p = y
5  x.p = y.p
6  if y.p == T.nil
7      T.root = x
8  elseif y == y.p.right
9      y.p.right = x
10 else y.p.left = x
11 x.right = y
12 y.p = x
    
```

13.2-2

Argue that in every n -node binary search tree, there are exactly $n - 1$ possible rotations.

Every node can rotate with its parent, only the root does not have a parent, therefore there are $n - 1$ possible rotations.

13.2-3

Let a , b , and c be arbitrary nodes in subtrees α , β , and γ , respectively, in the left tree of Figure 13.2. How do the depths of a , b , and c change when a left rotation is performed on node x in the figure?

a : increase by 1.

b : unchanged.

c : decrease by 1.

13.2-4

Show that any arbitrary n -node binary search tree can be transformed into any other arbitrary n -node binary search tree using $O(n)$ rotations.

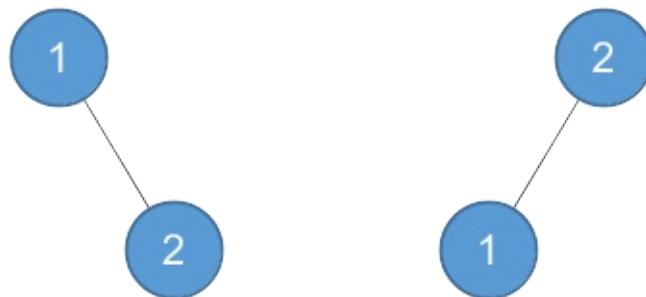
For each left leaf, we perform right rotation until there is no left leaf. We need at most $n - 1$ right rotations to transform the tree into a right-going chain.

Since we can transform every tree into a right-going chain, and the operation is invertible, therefore we can transform one tree into the right-going chain and use the inverse operation to construct the other tree.

13.2-5 *

We say that a binary search tree T_1 can be right-converted to binary search tree T_2 if it is possible to obtain T_2 from T_1 via a series of calls to RIGHT-ROTATE. Give an example of two trees T_1 and T_2 such that T_1 cannot be right-converted to T_2 . Then, show that if a tree T_1 can be right-converted to T_2 , it can be right-converted using $O(n^2)$ calls to RIGHT-ROTATE.

<https://github.com/CyberZH3/> <https://github.com/CyberZH3/>



We can use $O(n)$ calls to rotate the node which is the root in T_2 to T_1 's root, then use the same operation in the two subtrees. There are n nodes, therefore the upper bound is $O(n^2)$.

13.3 Insertion

13.3-1

In line 16 of RB-INSERT, we set the color of the newly inserted node z to red. Observe that if we had chosen to set z 's color to black, then property 4 of a red-black tree would not be violated. Why didn't we choose to set z 's color to black?

Violate property 5.

13.3-2

Show the red-black trees that result after successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty red-black tree.

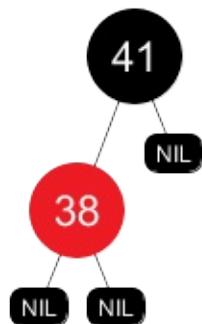
Insert 41:

<https://github.com/CyberZ/>



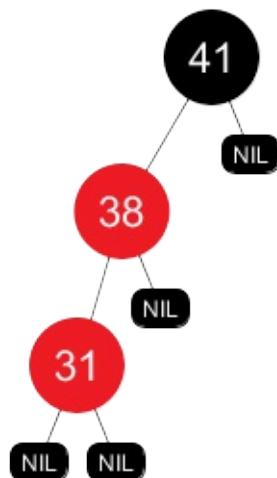
Insert 38:

<https://github.com/CyberZ/>

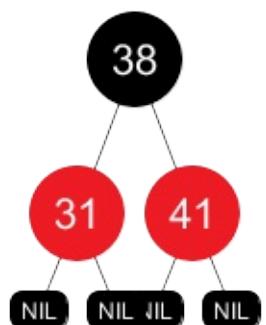


Insert 31:

<https://github.com/CyberZH6>

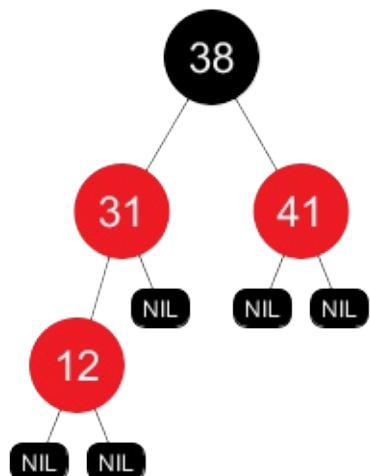


<https://github.com/CyberZH6>

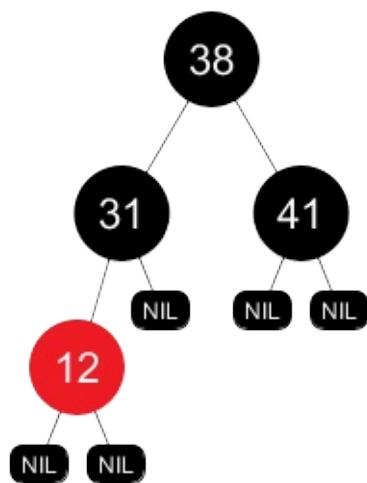


Insert 12:

<https://github.com/CyberZH6>

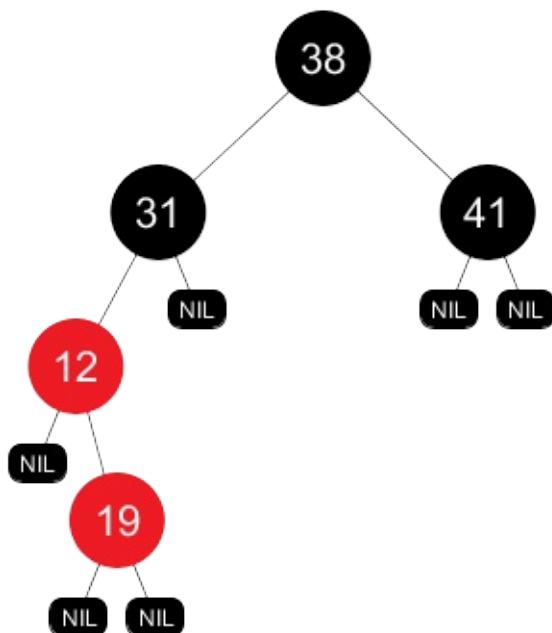


<https://github.com/CyberZH0/>

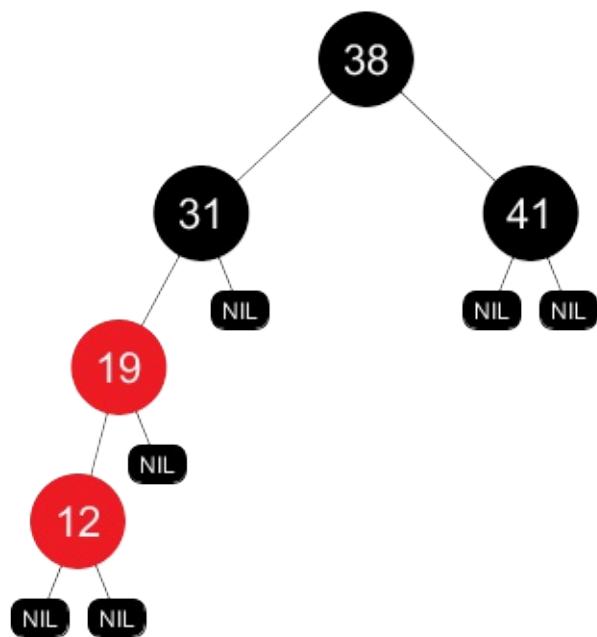


Insert 19:

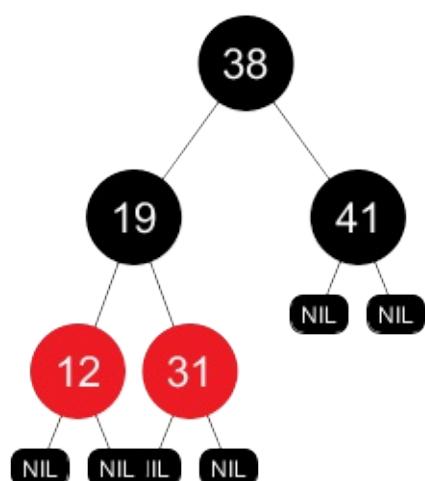
<https://github.com/CyberZH0/>



https://github.com/CyberZH/CS-Notes/blob/main/Algorithms/Binary%20Search%20Tree/insertion/insertion_1.png

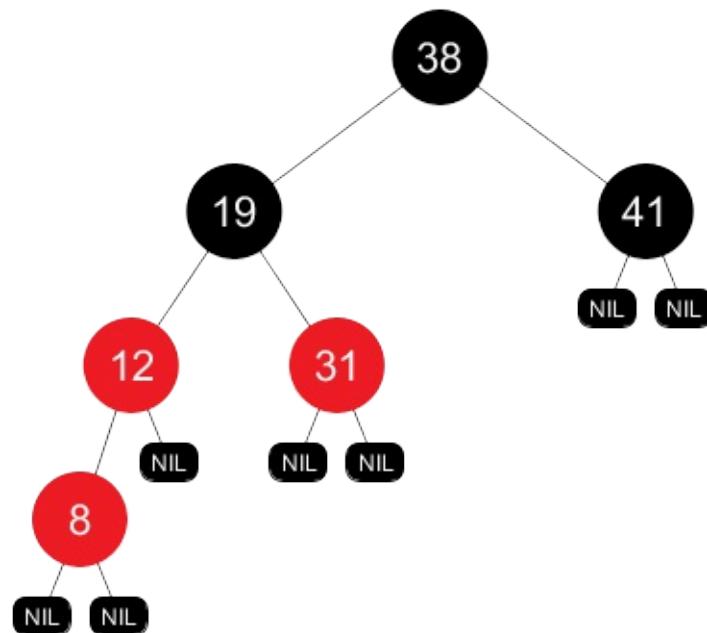


https://github.com/CyberZH/CS-Notes/blob/main/Algorithms/Binary%20Search%20Tree/insertion/insertion_2.png

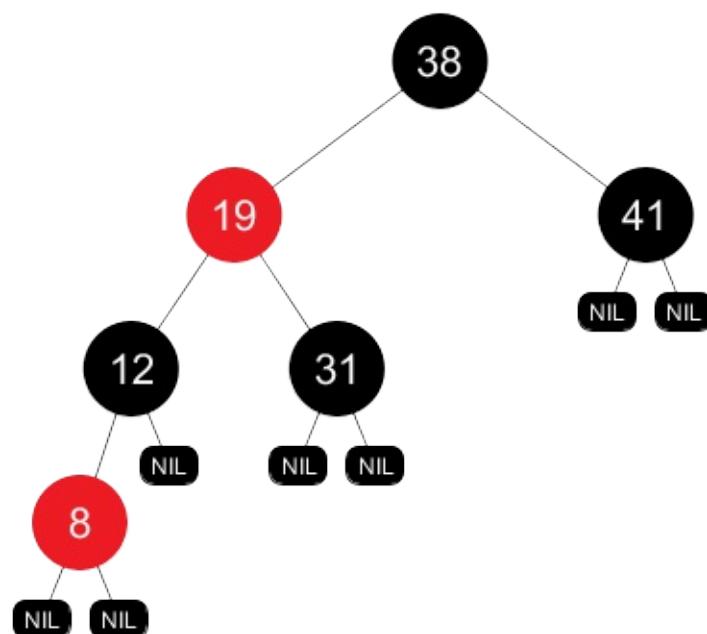


Insert 8:

https://github.com/CyberZH/CLRS-Python/blob/main/13_BST.ipynb

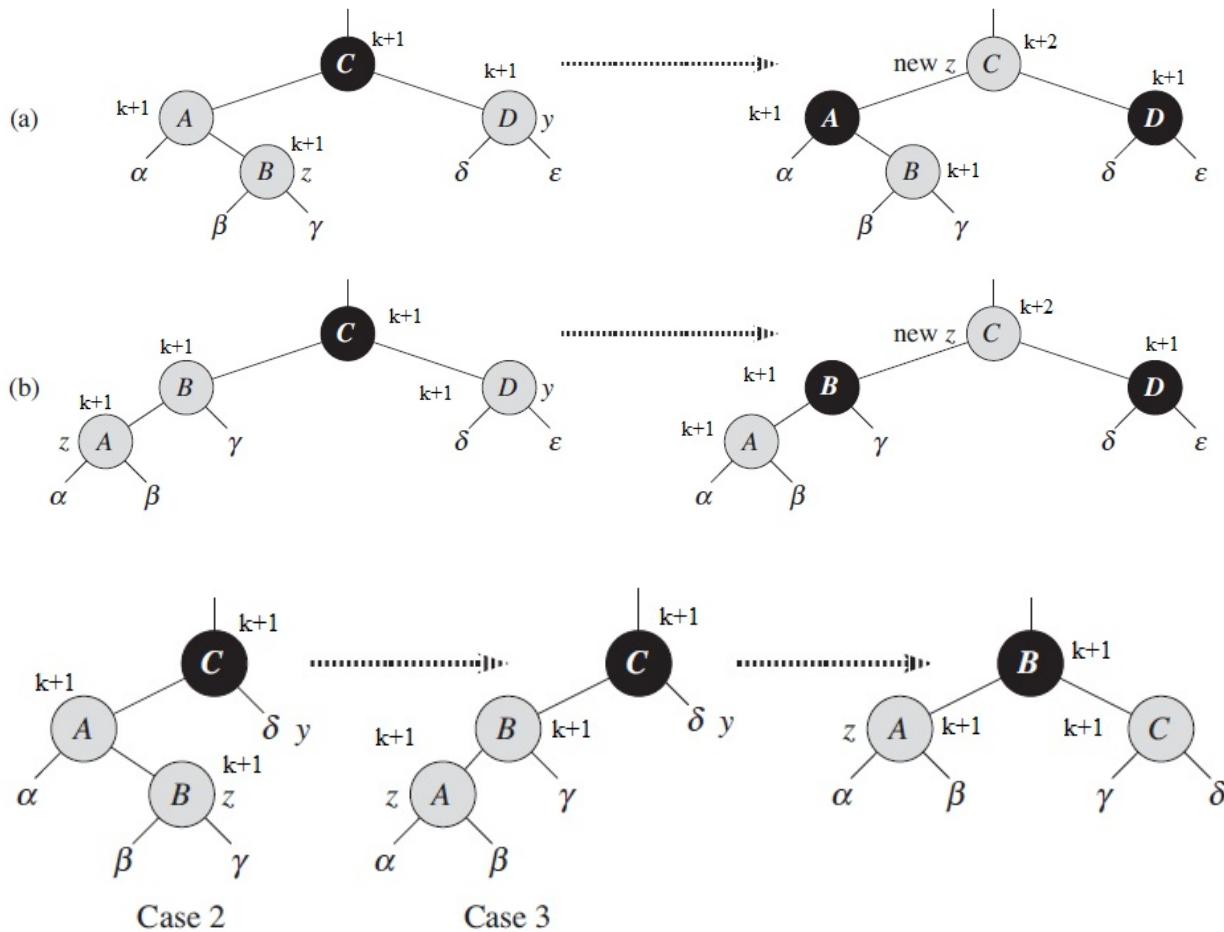


https://github.com/CyberZH/CLRS-Python/blob/main/13_BST.ipynb



13.3-3

Suppose that the black-height of each of the subtrees $\alpha, \beta, \gamma, \delta, \epsilon$ in Figures 13.5 and 13.6 is k . Label each node in each figure with its black-height to verify that the indicated transformation preserves property 5.



13.3-4

Professor Teach is concerned that RB-INSERT-FIXUP might set $T.\text{nil}.color$ to RED, in which case the test in line 1 would not cause the loop to terminate when z is the root. Show that the professor's concern is unfounded by arguing that RBINSERT-FIXUP never sets $T.\text{nil}.color$ to RED.

In order to set $T.\text{nil}.color$ to RED, $z.p$ must be the root; and if $z.p$ is the root, then $z.p$ is black, the loop terminates.

13.3-5

Consider a red-black tree formed by inserting n nodes with RB-INSERT. Argue that if $n > 1$, the tree has at least one red node.

In case 1, z and $z.p$ are RED, if the loop terminates, then z could not be the root, thus z is RED after the fix up.

In case 2, z and $z.p$ are RED, and after the rotation $z.p$ could not be the root, thus $z.p$ is RED after the fix up.

In case 3, z is RED and z could not be the root, thus z is RED after the fix up.

Therefore, there is always at least one red node.

13.3-6

Suggest how to implement RB-INSERT efficiently if the representation for red-black trees includes no storage for parent pointers.

Use stack to record the path to the inserted node, then parent is the top element in the stack.

In case 1, we pop $z \cdot p$ and $z \cdot p \cdot p$.

In case 2, we pop $z \cdot p$ and $z \cdot p \cdot p$, then push $z \cdot p \cdot p$ and z .

In case 3, we pop $z \cdot p$, $z \cdot p \cdot p$ and $z \cdot p \cdot p \cdot p$, then push $z \cdot p$.

```

RED = 0
BLACK = 1

class Stack:
    def __init__(self):
        self.vals = []

    def push(self, x):
        self.vals.append(x)

    def pop(self):
        if len(self.vals) == 0:
            return None
        x = self.vals[-1]
        del self.vals[-1]
        return x

class RedBlackTreeNode:
    def __init__(self, key, left=None, right=None):
        self.color = BLACK
        self.key = key
        self.left = left
        self.right = right

class RedBlackTree:
    def __init__(self):
        self.nil = RedBlackTreeNode(None)
        self.nil.color = BLACK
        self.nil.left = self.nil
        self.nil.right = self.nil

```

```

        self.root = self.nil

def left_rotate(T, x, p):
    y = x.right
    x.right = y.left
    if p == T.nil:
        T.root = y
    elif x == p.left:
        p.left = y
    else:
        p.right = y
    y.left = x

def right_rotate(T, x, p):
    y = x.left
    x.left = y.right
    if p == T.nil:
        T.root = y
    elif x == p.right:
        p.right = y
    else:
        p.left = y
    y.right = x

def rb_insert_fixup(T, z, stack):
    while True:
        p = stack.pop()
        if p.color == BLACK:
            break
        pp = stack.pop()
        if p == pp.left:
            y = pp.right
            if y.color == RED:
                p.color = BLACK
                y.color = BLACK
                pp.color = RED
                z = pp
            elif z == p.right:
                stack.push(pp)
                stack.push(z)
                z = p
                left_rotate(T, z, pp)
            else:
                ppp = stack.pop()
                stack.push(p)
                p.color = BLACK
                pp.color = RED
                right_rotate(T, pp, ppp)
        else:
            y = pp.left

```

```

    if y.color == RED:
        p.color = BLACK
        y.color = BLACK
        pp.color = RED
        z = pp
    elif z == p.left:
        stack.push(pp)
        stack.push(z)
        z = p
        right_rotate(T, z, pp)
    else:
        ppp = stack.pop()
        stack.push(p)
        p.color = BLACK
        pp.color = RED
        left_rotate(T, pp, ppp)
T.root.color = BLACK

def rb_insert(T, z):
    stack = Stack()
    stack.push(T.nil)
    y = T.nil
    x = T.root
    while x != T.nil:
        stack.push(x)
        y = x
        if z.key < x.key:
            x = x.left
        else:
            x = x.right
    if y == T.nil:
        T.root = z
    elif z.key < y.key:
        y.left = z
    else:
        y.right = z
    z.left = T.nil
    z.right = T.nil
    z.color = RED
    rb_insert_fixup(T, z, stack)

```

13.4 Deletion

13.4-1

Argue that after executing RB-DELETE-FIXUP, the root of the tree must be black.

Case 1, transform to 2, 3, 4.

Case 2, if terminates, the root of the subtree (the new x) is set to black.

Case 3, transform to 4.

Case 4, the root (the new x) is set to black.

13.4-2

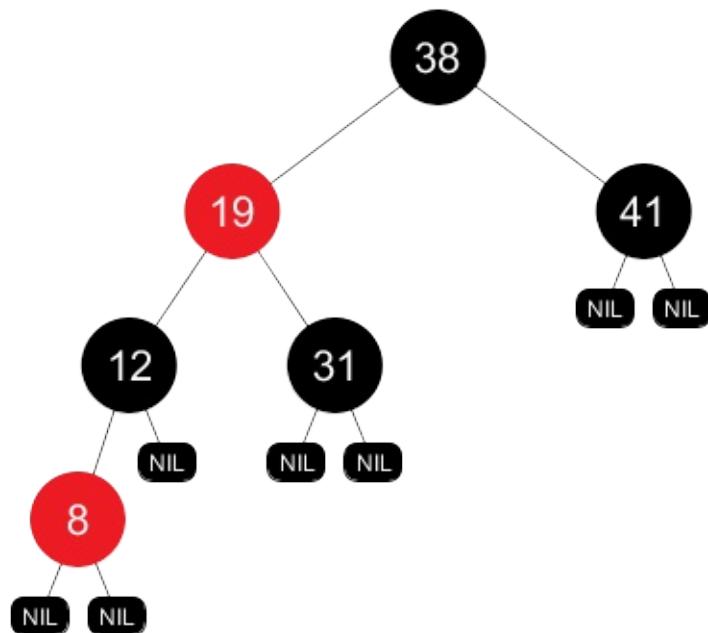
Argue that if in RB-DELETE both x and $x.P$ are red, then property 4 is restored by the call to RB-DELETE-FIXUP (T, x) .

Will not enter the loop, x is set to black.

13.4-3

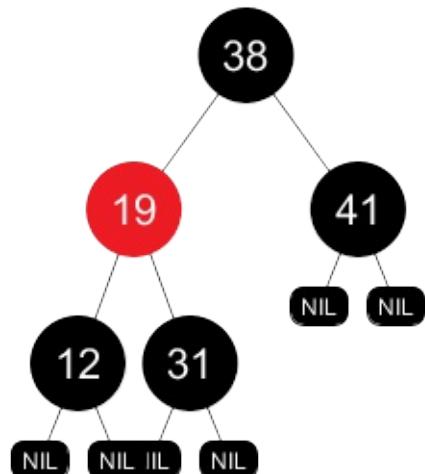
In Exercise 13.3-2, you found the red-black tree that results from successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty tree. Now show the red-black trees that result from the successive deletion of the keys in the order 8, 12, 19, 31, 38, 41.

<https://github.com/CyberZH/CS-Notes/blob/main/Notes/13.%20AVL%20Tree/13.4%20Deletion.md>



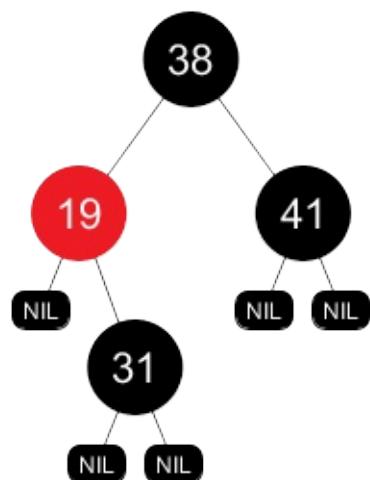
Delete 8:

<https://github.com/CyberZH/CS-Notes/blob/main/Notes/13.%20AVL%20Tree/13.4%20Deletion%20-%20Delete%208.md>

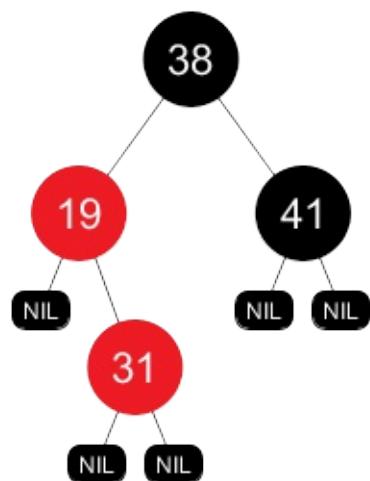


Delete 12:

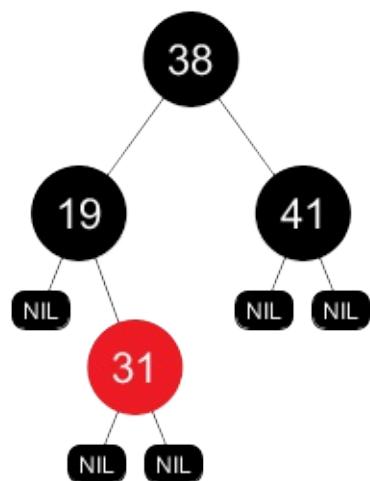
<https://github.com/CyberZH5/>



<https://github.com/CyberZH5/>

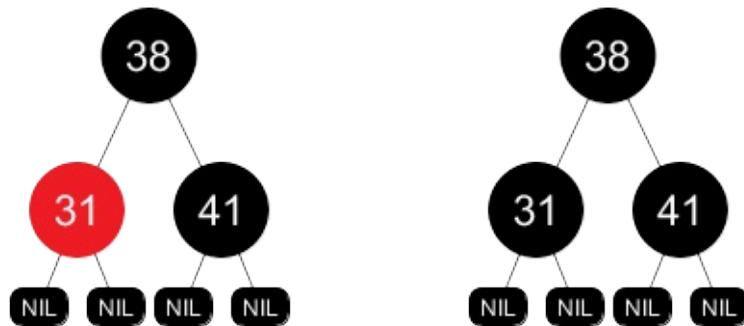


<https://github.com/CyberZH5/>



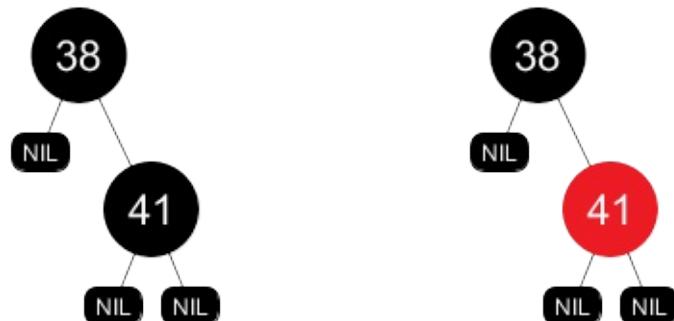
Delete 19:

<https://github.com/CyberZHG> <https://github.com/CyberZHG>



Delete 31:

<https://github.com/CyberZHG> <https://github.com/CyberZHG>



Delete 38:

<https://github.com/CyberZHG> <https://github.com/CyberZHG>



Delete 41.

13.4-4

In which lines of the code for RB-DELETE-FIXUP might we examine or modify the sentinel $T.nil$?

Line 1, 2.

13.4-5

In each of the cases of Figure 13.7, give the count of black nodes from the root of the subtree shown to each of the subtrees $\alpha, \beta, \dots, \zeta$, and verify that each count remains the same after the transformation. When a node has a color attribute c or c' , use the notation $\text{count}(c)$ or $\text{count}(c')$ symbolically in your count.

...

13.4-6

Professors Skelton and Baron are concerned that at the start of case 1 of RBDELETE-FIXUP, the node $x.p$ might not be black. If the professors are correct, then lines 5–6 are wrong. Show that $x.p$ must be black at the start of case 1, so that the professors have nothing to worry about.

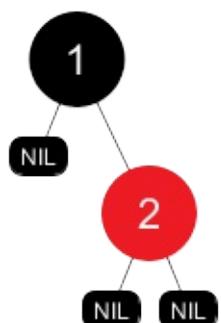
Since w is red, based on property 4, $x.p$ must be black.

13.4-7

Suppose that a node x is inserted into a red-black tree with RB-INSERT and then is immediately deleted with RB-DELETE. Is the resulting red-black tree the same as the initial red-black tree? Justify your answer.

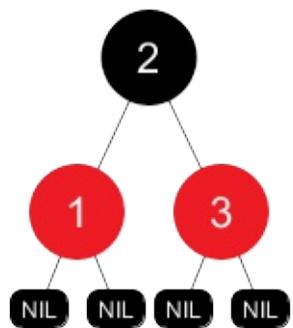
No.

<https://github.com/CyberZhi>



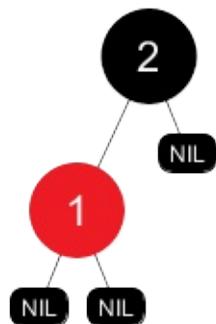
Insert 3:

<https://github.com/CyberZHG>



Delete 3:

<https://github.com/CyberZHG>



Problems

13-1 Persistent dynamic sets

During the course of an algorithm, we sometimes find that we need to maintain past versions of a dynamic set as it is updated. We call such a set ***persistent***. One way to implement a persistent set is to copy the entire set whenever it is modified, but this approach can slow down a program and also consume much space. Sometimes, we can do much better.

- a. For a general persistent binary search tree, identify the nodes that we need to change to insert a key k or delete a node y .

Insert: the number of nodes in the simple path plus 1.

Delete: the ancestors of y .

- b. Write a procedure PERSISTENT-TREE-INSERT that, given a persistent tree T and a key k to insert, returns a new persistent tree T' that is the result of inserting k into T

```
class TreeNode:
    def __init__(self, key, left=None, right=None):
        self.key = key
        self.left = left
        self.right = right

    def insert(root, x):
        if root is None:
            return TreeNode(x)
        new_root = TreeNode(root.key)
        if root.key <= x:
            new_root.left = root.left
            new_root.right = insert(root.right, x)
        else:
            new_root.left = insert(root.left, x)
            new_root.right = root.right
        return new_root
```

- c. If the height of the persistent binary search tree T is h , what are the time and space requirements of your implementation of PERSISTENT-TREE-INSERT?

$\Theta(h)$ and $\Theta(h)$.

d. Suppose that we had included the parent attribute in each node. In this case, PERSISTENT-TREE-INSERT would need to perform additional copying. Prove that PERSISTENT-TREE-INSERT would then require $\Omega(n)$ time and space, where n is the number of nodes in the tree.

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

e. Show how to use red-black trees to guarantee that the worst-case running time and space are $O(\lg n)$ per insertion or deletion.

Based on Exercise 13.3-6.

13-2 Join operation on red-black trees

The **join** operation takes two dynamic sets S_1 and S_2 and an element x such that for any $x_1 \in S_1$ and $x_2 \in S_2$, we have $x_1.key \leq x.key \leq x_2.key$. It returns a set $S = S_1 \cup \{x\} \cup S_2$. In this problem, we investigate how to implement the join operation on red-black trees.

a. Given a red-black tree T , let us store its black-height as the new attribute $T.bh$. Argue that RB-INSERT and RB-DELETE can maintain the bh attribute without requiring extra storage in the nodes of the tree and without increasing the asymptotic running times. Show that while descending through T , we can determine the black-height of each node we visit in $O(1)$ time per node visited.

Initialize: $bh = 0$.

RB-INSERT: if in the last step the root is red, we increase bh by 1.

RB-DELETE: if x is root, we decrease bh by 1.

Each node: in the simple path, decrease bh by 1 each time we find a black node.

We wish to implement the operation RB-JOIN (T_1, x, T_2) , which destroys T_1 and T_2 and returns a red-black tree $T = T_1 \cup \{x\} \cup T_2$. Let n be the total number of nodes in T_1 and T_2 .

b. Assume that $T_1.bh \geq T_2.bh$. Describe an $O(\lg n)$ -time algorithm that finds a black node y in T_1 with the largest key from among those nodes whose black-height is $T_2.bh$.

Move to the right child if the node has a right child, otherwise move to the left child. If the node is black, we decrease bh by 1. Repeat the step until $bh = T2.bh$.

c. Let T_y be the subtree rooted at y . Describe how $T_y \cup \{x\} \cup T_2$ can replace T_y in $O(1)$ time without destroying the binary-search-tree property.

x 's parent is T_y 's parent, x 's left child is T_y and its right child is T_2 .

d. What color should we make x so that red-black properties 1, 3, and 5 are maintained? Describe how to enforce properties 2 and 4 in $O(\lg n)$ time.

Red. RB-INSERT-FIXUP(T, x).

e. Argue that no generality is lost by making the assumption in part (b). Describe the symmetric situation that arises when $T_1.bh \leq T_2.bh$.

Symmetric.

f. Argue that the running time of RB-JOIN is $O(\lg n)$.

$$O(1) + O(\lg n) = O(\lg n)$$

13-3 AVL trees

An **AVL tree** is a binary search tree that is **height balanced**: for each node x , the heights of the left and right subtrees of x differ by at most 1. To implement an AVL tree, we maintain an extra attribute in each node: $x.h$ is the height of node x . As for any other binary search tree T , we assume that $T.root$ points to the root node.

a. Prove that an AVL tree with n nodes has height $O(\lg n)$.

$$T(h) = T(h - 1) + T(h - 2)$$

b. To insert into an AVL tree, we first place a node into the appropriate place in binary search tree order. Afterward, the tree might no longer be height balanced. Specifically, the heights of the left and right children of some node might differ by 2. Describe a procedure $BALANCE(x)$, which takes a subtree rooted at x whose left and right children are height balanced and have heights that differ by at most 2, i.e., $|x.right.h - x.left.h| \leq 2$, and alters the subtree rooted at x to be height balanced.

See c.

- c. Using part (b), describe a recursive procedure $\text{AVL-INSERT } (x, z)$ that takes a node x within an AVL tree and a newly created node z (whose key has already been filled in), and adds z to the subtree rooted at x , maintaining the property that x is the root of an AVL tree. As in TREE-INSERT from Section 12.3, assume that $z.\text{key}$ has already been filled in and that $z.\text{left} = \text{NIL}$ and $z.\text{right} = \text{NIL}$; also assume that $z.h = 0$. Thus, to insert the node z into the AVL tree T , we call $\text{AVL-INSERT}(T.\text{root}, z)$.

```

class AVLTreeNode:
    def __init__(self, key, left=None, right=None):
        self.key = key
        self.h = 0
        self.p = None
        self.left = left
        self.right = right
        if self.left is not None:
            self.left.p = self
        if self.right is not None:
            self.right.p = self

class AVL:
    def __init__(self):
        self.root = None

    def left_rotate(self, x):
        y = x.right
        x.right = y.left
        if y.left is not None:
            y.left.p = x
        y.p = x.p
        if x.p is None:
            self.root = y
        elif x == x.p.left:
            x.p.left = y
        else:
            x.p.right = y
        y.left = x
        x.p = y

    def right_rotate(self, x):
        y = x.left
        x.left = y.right
        if y.right is not None:
            y.right.p = x
        y.p = x.p
        if x.p is None:
            self.root = y
        elif x == x.p.left:

```

```
x.p.left = y
else:
    x.p.right = y
y.right = x
x.p = y

def get_height(self, node):
    if node is None:
        return -1
    return node.h

def update_height(self, node):
    if node is None:
        return
    node.h = max(self.get_height(node.left), self.get_height(node.right))+1

def balance_factor(self, node):
    return self.get_height(node.left) - self.get_height(node.right)

def avl_insert(self, x):
    self.root = self.avl_insert_rec(self.root, x)

def avl_insert_rec(self, root, x):
    if root is None:
        return AVLTreeNode(x)
    if root.key > x:
        root.left = self.avl_insert_rec(root.left, x)
        root.left.p = root
    else:
        root.right = self.avl_insert_rec(root.right, x)
        root.right.p = root
    if self.balance_factor(root) == 2:
        if self.balance_factor(root.left) == -1:
            self.left_rotate(root.left)
            self.right_rotate(root)
        root = root.p
        self.update_height(root.left)
        self.update_height(root.right)
        self.update_height(root)
    elif self.balance_factor(root) == -2:
        if self.balance_factor(root.right) == 1:
            self.right_rotate(root.right)
            self.left_rotate(root)
        root = root.p
        self.update_height(root.left)
        self.update_height(root.right)
        self.update_height(root)
    else:
        self.update_height(root)
    return root
```

- d.** Show that AVL-INSERT, run on an n -node AVL tree, takes $O(\lg n)$ time and performs $O(1)$ rotations.

$O(\lg n)$: the length of path from root to the inserted node.

$O(1)$: the height will decrease by 1 after the rotation, therefore the ancestors will not be affected.

13-4 Treaps

If we insert a set of n items into a binary search tree, the resulting tree may be horribly unbalanced, leading to long search times. As we saw in Section 12.4, however, randomly built binary search trees tend to be balanced. Therefore, one strategy that, on average, builds a balanced tree for a fixed set of items would be to randomly permute the items and then insert them in that order into the tree.

- a.** Show that given a set of nodes x_1, x_2, \dots, x_n , with associated keys and priorities, all distinct, the treap associated with these nodes is unique.

The root is the node with smallest priority, the root divides the sets into two subsets based on the key. In each subset, the node with smallest priority is selected as the root, thus we can uniquely determine a treap with a specific input.

- b.** Show that the expected height of a treap is $\Theta(\lg n)$, and hence the expected time to search for a value in the treap is $\Theta(\lg n)$.

Same as randomly built BST.

- c.** Explain how TREAP-INSERT works. Explain the idea in English and give pseudocode.

```
class TreapNode:
    def __init__(self, key, left=None, right=None):
        self.key = key
        self.priority = random.random()
        self.p = None
        self.left = left
        self.right = right
        if self.left is not None:
            self.left.p = self
        if self.right is not None:
            self.right.p = self

class Treap:
```

```
def __init__(self):
    self.root = None

def left_rotate(self, x):
    y = x.right
    x.right = y.left
    if y.left is not None:
        y.left.p = x
    y.p = x.p
    if x.p is None:
        self.root = y
    elif x == x.p.left:
        x.p.left = y
    else:
        x.p.right = y
    y.left = x
    x.p = y

def right_rotate(self, x):
    y = x.left
    x.left = y.right
    if y.right is not None:
        y.right.p = x
    y.p = x.p
    if x.p is None:
        self.root = y
    elif x == x.p.left:
        x.p.left = y
    else:
        x.p.right = y
    y.right = x
    x.p = y

def insert(self, x):
    self.root = self.insert_rec(self.root, x)

def insert_rec(self, root, x):
    if root is None:
        return TreapNode(x)
    if root.key > x:
        root.left = self.insert_rec(root.left, x)
        root.left.p = root
        if root.left.priority < root.priority:
            self.right_rotate(root)
            root = root.p
    else:
        root.right = self.insert_rec(root.right, x)
        root.right.p = root
        if root.right.priority < root.priority:
            self.left_rotate(root)
            root = root.p
    return root
```

d. Show that the expected running time of TREAP-INSERT is $\Theta(\lg n)$.

Rotation is $\Theta(1)$, at most h rotations, therefore the expected running time is $\Theta(\lg n)$.

e. Consider the treap T immediately after TREAP-INSERT has inserted node x . Let C be the length of the right spine of the left subtree of x . Let D be the length of the left spine of the right subtree of x . Prove that the total number of rotations that were performed during the insertion of x is equal to $C + D$.

Left rotation increase C by 1, right rotation increase D by 1.

f. Show that $X_{ik} = 1$ if and only if $y.\text{priority} > x.\text{priority}$, $y.\text{key} < x.\text{key}$, and, for every z such that $y.\text{key} < z.\text{key} < x.\text{key}$, we have $y.\text{priority} < z.\text{priority}$.

The first two are obvious.

The min-heap property will not hold if $y.\text{priority} > z.\text{priority}$.

g. Show that

$$\begin{aligned}\Pr\{X_{ik} = 1\} &= \frac{(k-i-1)!}{(k-i+1)!} \\ &= \frac{1}{(k-i+1)(k-i)}\end{aligned}$$

Total number of permutations: $(k-i+1)!$

Permutations satisfy the condition: $(k-i-1)!$

h. Show that

$$\begin{aligned}\mathbb{E}[C] &= \sum_{j=1}^{k-1} \frac{1}{j(j+1)} \\ &= 1 - \frac{1}{k}\end{aligned}$$

$$\begin{aligned}
 \mathbb{E}[C] &= \sum_{j=1}^{k-1} \frac{1}{(k-i+1)(k-i)} \\
 &= \sum_{j=1}^{k-1} \left(\frac{1}{k-i} - \frac{1}{k-i+1} \right) \\
 &= 1 - \frac{1}{k}
 \end{aligned}$$

i. Use a symmetry argument to show that

$$\mathbb{E}[D] = 1 - \frac{1}{n-k+1}$$

$$\begin{aligned}
 \mathbb{E}[D] &= \sum_{j=1}^{n-k} \frac{1}{(k-i+1)(k-i)} \\
 &= 1 - \frac{1}{n-k+1}
 \end{aligned}$$

j. Conclude that the expected number of rotations performed when inserting a node into a treap is less than 2.

$$\mathbb{E}[C] + \mathbb{E}[D] \leq 2$$

14 Augmenting Data Structures

- 14.1 Dynamic order statistics
- 14.2 How to augment a data structure
- 14.3 Interval trees
- Problems

14.1 Dynamic order statistics

14.1-1

Show how OS-SELECT ($T.root, 10$) operates on the red-black tree T of Figure 14.1.

- 26: $r = 13, i = 10$, go left
- 17: $r = 8, i = 10$, go right
- 21: $r = 3, i = 2$, go left
- 19: $r = 1, i = 2$, go right
- 20: $r = 1, i = 1$, choose 20

14.1-2

Show how OS-RANK (T, x) operates on the red-black tree T of Figure 14.1 and the node x with $x.key = 35$.

- 35: $r = 1$
- 38: $r = 1$
- 30: $r = r + 2 = 3$
- 41: $r = 3$
- 26: $r = r + 13 = 16$

14.1-3

Write a nonrecursive version of OS-SELECT.

```

class TreeNode:
    def __init__(self, key, left=None, right=None):
        self.key = key
        self.size = 1
        self.left = left
        self.right = right
        if left is not None:
            self.size += left.size
        if right is not None:
            self.size += right.size

def os_select(x, i):
    while True:
        if x.left is None:
            r = 1
        else:
            r = x.left.size + 1
        if i == r:
            return x
        elif i < r:
            x = x.left
        else:
            x = x.right
            i -= r

```

14.1-4

Write a recursive procedure $\text{OS-KEY-RANK } (T, k)$ that takes as input an order-statistic tree T and a key k and returns the rank of k in the dynamic set represented by T . Assume that the keys of T are distinct.

```

class TreeNode:
    def __init__(self, key, left=None, right=None):
        self.key = key
        self.size = 1
        self.left = left
        self.right = right
        if left is not None:
            self.size += left.size
        if right is not None:
            self.size += right.size

def os_key_rank(x, k, i=0):
    r = 1
    if x.left is not None:
        r += x.left.size
    if k == x.key:
        return i + r
    if k < x.key:
        return os_key_rank(x.left, k, i)
    if k > x.key:
        return os_key_rank(x.right, k, i + r)

```

14.1-5

Given an element x in an n -node order-statistic tree and a natural number i , how can we determine the i th successor of x in the linear order of the tree in $O(\lg n)$ time?

$\text{OS-SELECT}(T, \text{OS-RANK}(T, x) + i)$

14.1-6

Observe that whenever we reference the size attribute of a node in either OS-SELECT or OS-RANK, we use it only to compute a rank. Accordingly, suppose we store in each node its rank in the subtree of which it is the root. Show how to maintain this information during insertion and deletion. (Remember that these two operations can cause rotations.)

Tree walk and change the rank by comparing the key of the node with that of the inserted node. $O(n)$

14.1-7

Show how to use an order-statistic tree to count the number of inversions (see Problem 2-4) in an array of size n in time $O(n \lg n)$.

After the insertion of a node, the number of tree nodes subtract the rank of the inserted node is the number of inversions of the current node.

14.1-8 *

Consider n chords on a circle, each defined by its endpoints. Describe an $O(n \lg n)$ - time algorithm to determine the number of pairs of chords that intersect inside the circle. (For example, if the n chords are all diameters that meet at the center, then the correct answer is $\binom{n}{2}$. Assume that no two chords share an endpoint.

Sort the vertices in clock-wise order, and assign a unique value to each vertex. For each chord its two vertices are u_i, v_i and $u_i < v_i$. Add the vertices one by one in clock-wise order, if we meet a u_i , we add it to the order-statistic tree, if we meet a v_i , we calculate how many nodes are larger than u_i (which is the number of intersects with chord i), and remove u_i .

14.2 How to augment a data structure

14.2-1

Show, by adding pointers to the nodes, how to support each of the dynamic-set queries MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR in $O(1)$ worstcase time on an augmented order-statistic tree. The asymptotic performance of other operations on order-statistic trees should not be affected.

MINIMUM: A pointer points to the minimum node, if the node is being deleted, move the pointer to its successor.

MAXIMUM: Similar to MINIMUM.

SUCCESSOR: Every node records its successor, the insertion and deletion is similar to that in linked list.

PREDECESSOR: Similar to MAXIMUM.

14.2-2

Can we maintain the black-heights of nodes in a red-black tree as attributes in the nodes of the tree without affecting the asymptotic performance of any of the redblack tree operations? Show how, or argue why not. How about maintaining the depths of nodes?

$$x.h = \max(x.left.h, y.left.h) + 1$$

14.2-3 *

Let \otimes be an associative binary operator, and let a be an attribute maintained in each node of a red-black tree. Suppose that we want to include in each node x an additional attribute f such that $x.f = x_1.a \otimes x_2.a \otimes \dots \otimes x_m.a$, where x_1, x_2, \dots, x_m is the inorder listing of nodes in the subtree rooted at x . Show how to update the f attributes in $O(1)$ time after a rotation. Modify your argument slightly to apply it to the size attributes in order-statistic trees.

$$x.f = x.left.f \otimes x.a \otimes x.right.f$$

14.2-4 *

We wish to augment red-black trees with an operation RB ENUMERATE (x, a, b) that outputs all the keys k such that $a \leq k \leq b$ in a red-black tree rooted at x . Describe how to implement RB ENUMERATE in $\Theta(m + \lg n)$ time, where m is the number of keys that are output and n is the number of internal nodes in the tree.

$\Theta(\lg n)$: Find the smallest key that larger than or equal to a .

$\Theta(m)$: Based on Exercise 14.2-1, find the m successor.

14.3 Interval trees

14.3-1

Write pseudocode for LEFT-ROTATE that operates on nodes in an interval tree and updates the \max attributes in $O(1)$ time.

$y.\max = x.\max$

$x.\max = \max(x.\text{int}.high, x.left.\max, x.right.\max)$

14.3-2

Rewrite the code for INTERVAL-SEARCH so that it works properly when all intervals are open.

```
INTERVAL-SEARCH(T, i)
1 x = T.root
2 while x != T.nil and (i.high <= x.int.left or x.int.right <= i.low)
3     if x.left != T.nil and x.left.max > i.low
4         x = x.left
5     else x = x.right
6 return x
```

14.3-3

Describe an efficient algorithm that, given an interval i , returns an interval overlapping i that has the minimum low endpoint, or $T.\text{nil}$ if no such interval exists.

```
MIN-INTERVAL-SEARCH(T, i)
1 x = T.root
2 ret = T.nil
3 while x != T.nil:
4     if not (i.high <= x.int.left or x.int.right <= i.low)
5         if ret == T.nil or ret.right > x.int.right
6             ret = x
7         if x.left != T.nil and x.left.max > i.low
8             x = x.left
9         else x = x.right
10 return ret
```

14.3-4

Given an interval tree T and an interval i , describe how to list all intervals in T that overlap i in $O(\min(n, k \lg n))$ time, where k is the number of intervals in the output list.

```

INTERVALS-SEARCH(T, x, i)
1 lst = []
2 if i overlaps x.int
3     lst.append(x)
4 if x.left != T.nil and x.left.max > i.low
5     lst += INTERVALS-SEARCH(T, x.left, i)
6 if x.right != T.nil and x.int.low <= i.high and x.right.max >= i.low
7     lst += INTERVALS-SEARCH(T, x.right, i)
8 return lst

```

14.3-5

Suggest modifications to the interval-tree procedures to support the new operation INTERVAL-SEARCH-EXACTLY (T, i) , where T is an interval tree and i is an interval. The operation should return a pointer to a node x in T such that $x.int.low = i.low$ and $x.int.high = i.high$, or $T.nil$ if T contains no such node. All operations, including INTERVAL-SEARCH-EXACTLY, should run in $O(\lg n)$ time on an n -node interval tree.

Search for nodes which has exactly the same low value.

14.3-6

Show how to maintain a dynamic set Q of numbers that supports the operation MIN-GAP, which gives the magnitude of the difference of the two closest numbers in Q . For example, if $Q = \{1, 5, 9, 15, 18, 22\}$, then MIN-GAP (Q) returns $18 - 15 = 3$, since 15 and 18 are the two closest numbers in Q . Make the operations INSERT, DELETE, SEARCH, and MIN-GAP as efficient as possible, and analyze their running times.

Based on Exercise 14.2-1, we can maintain SUCCESSOR in $O(1)$ time, each time after updating the SUCCESSOR, we can update $x.gap$ to $x.successor.key - x.key$. And based on Exercise 14.2-1 we can also maintain the minimum gap of the subtree in $O(1)$ time.

14.3-7 *

VLSI databases commonly represent an integrated circuit as a list of rectangles. Assume that each rectangle is rectilinearly oriented (sides parallel to the x - and y -axes), so that we represent a rectangle by its minimum and maximum x and y -coordinates. Give an $O(n \lg n)$ -time algorithm to decide whether or not a set of n rectangles so represented contains two rectangles that overlap. Your algorithm need not report all intersecting pairs, but it must report that an overlap exists if one rectangle entirely covers another, even if the boundary lines do not intersect.

Suppose we represent a rectangle by $(x_{min}, x_{max}, y_{min}, y_{max})$.

Sort the x_{min} s and x_{max} s in ascending order. From left to right, if we meet a x_{min} , before we add (y_{min}, y_{max}) to the interval tree, if the interval (y_{min}, y_{max}) is overlapped with some node in the interval tree, then there is an overlap of rectangles. And when we meet a x_{max} , we remove (y_{min}, y_{max}) from the interval tree.

Problems

14-1 Point of maximum overlap

Suppose that we wish to keep track of a point of maximum overlap in a set of intervals - a point with the largest number of intervals in the set that overlap it.

- a. Show that there will always be a point of maximum overlap that is an endpoint of one of the segments.
- b. Design a data structure that efficiently supports the operations INTERVAL-INSERT, INTERVAL-DELETE, and FIND-POM, which returns a point of maximum overlap.

14-2 Josephus permutation

We define the **Josephus problem** as follows. Suppose that n people form a circle and that we are given a positive integer $m \geq n$. Beginning with a designated first person, we proceed around the circle, removing every m th person. After each person is removed, counting continues around the circle that remains. This process continues until we have removed all n people. The order in which the people are removed from the circle defines the (n, m) -**Josephus permutation** of the integers $1, 2, \dots, n$. For example, the $(7, 3)$ -Josephus permutation is $\langle 3, 6, 2, 7, 5, 1, 4 \rangle$.

- a. Suppose that m is a constant. Describe an $O(n)$ -time algorithm that, given an integer n , outputs the (n, m) -Josephus permutation.

Use doubly linked list, the time is $O(nm)$, since m is a constant, $O(nm) = O(n)$.

```

class LinkedListNode:
    def __init__(self, key):
        self.key = key
        self.prev = None
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def insert(self, key):
        x = LinkedListNode(key)
        if self.head is None:
            self.head = x
            x.next = x
            x.prev = x
        else:
            x.prev = self.head.prev
            x.next = self.head
            x.prev.next = x
            x.next.prev = x

    def remove(self):
        if self.head.next == self.head:
            self.head = None
        else:
            self.head.next.prev = self.head.prev
            self.head.prev.next = self.head.next
            self.head = self.head.next

    def forward(self, step):
        while step > 0:
            step -= 1
            self.head = self.head.next

def josephus_permutation(n, m):
    lst = LinkedList()
    for i in xrange(1, n + 1):
        lst.insert(i)
    perm = []
    while lst.head is not None:
        lst.forward(m - 1)
        perm.append(lst.head.key)
        lst.remove()
    return perm

```

- b.** Suppose that m is not a constant. Describe an $O(n \lg n)$ -time algorithm that, given integers n and m , outputs the (n, m) -Josephus permutation.

Build a balanced binary search tree in $O(n \lg n)$, maintain $size$ to support order-statistics. In each iteration, we select and delete the $(r + m - 1) \bmod T.root.size + 1$ th element.

```

class TreeNode:
    def __init__(self, key, left=None, right=None):
        self.key = key
        self.color = BLACK
        self.size = 1
        self.p = None
        self.left = left
        self.right = right
        if left is not None:
            left.p = self
            self.size += left.size
        if right is not None:
            right.p = self
            self.size += right.size

class BinarySearchTree:
    def __init__(self, a):
        self.root = self.build(a, 0, len(a))

    def build(self, a, l, r):
        if l >= r:
            return None
        mid = (l + r) // 2
        return TreeNode(a[mid], self.build(a, l, mid), self.build(a, mid+1, r))

    def get_size(self, x):
        if x is None:
            return 0
        return x.size

    def update_size(self, x):
        if x is not None:
            x.size = 1 + self.get_size(x.left) + self.get_size(x.right)

    def select(self, x, i):
        r = self.get_size(x.left) + 1
        if i == r:
            return x
        elif i < r:
            return self.select(x.left, i)
        else:
            return self.select(x.right, i - r)

    def minimum(self, x):
        while x.left is not None:
            x = x.left

```

```

        return x

def transplant(self, u, v):
    if u.p is None:
        self.root = v
    elif u == u.p.left:
        u.p.left = v
    else:
        u.p.right = v
    if v is not None:
        v.p = u.p

def delete(self, z):
    if z.left is None:
        self.transplant(z, z.right)
    elif z.right is None:
        self.transplant(z, z.left)
    else:
        y = self.minimum(z.right)
        p = y.p
        if y.p != z:
            self.transplant(y, y.right)
            y.right = z.right
            y.right.p = y
        self.transplant(z, y)
        y.left = z.left
        y.left.p = y
        while p != z and p != y:
            self.update_size(p)
            p = p.p
        self.update_size(y)
    while z.p is not None:
        z = z.p
        self.update_size(z)

def josephus_permutation(n, m):
    tree = BinarySearchTree(range(1, n + 1))
    perm = []
    rank = 0
    while n > 0:
        rank = (rank + m - 1) % n
        x = tree.select(tree.root, rank + 1)
        perm.append(x.key)
        tree.delete(x)
        n -= 1
    return perm

```

15 Dynamic Programming

- 15.1 Rod cutting
- 15.2 Matrix-chain multiplication
- 15.3 Elements of dynamic programming
- 15.4 Longest common subsequence
- 15.5 Optimal binary search trees
- Problems

15.1 Rod cutting

15.1-1

Show that equation (15.4) follows from equation (15.3) and the initial condition

$$T(0) = 1$$

For $n = 0$, $T(0) = 2^0 = 1$.

Suppose $T(i) = 2^i$ for i in $[0, n - 1]$, then

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) = 1 + 1 + 2 + 2^2 + \cdots + 2^{n-1} = 2^n - 1 + 1 = 2^n$$

15.1-2

Show, by means of a counterexample, that the following "greedy" strategy does not always determine an optimal way to cut rods. Define the density of a rod of length i to be p_i/i , that is, its value per inch. The greedy strategy for a rod of length n cuts off a first piece of length i , where $1 \leq i \leq n$, having maximum density. It then continues by applying the greedy strategy to the remaining piece of length $n - i$.

Suppose $p_1 = 1, p_2 = 8, p_3 = 14, p_4 = 0$, the densities

$p_1/1 = 1, p_2/4 = 2, p_3/3 = 4\frac{2}{3}$, for $n = 4$, the greedy result is 3 and 1, the total value is 15, and the dynamic programming solution is 2 and 2, which is 16.

15.1-3

Consider a modification of the rod-cutting problem in which, in addition to a price p_i for each rod, each cut incurs a fixed cost of c . The revenue associated with a solution is now the sum of the prices of the pieces minus the costs of making the cuts. Give a dynamic-programming algorithm to solve this modified problem.

$$r_n = \max(\max_{1 \leq i \leq n-1}(p_i + r_{n-i}) - c, p_n)$$

```
def cut_rod(p, n, c):
    r = [0 for _ in xrange(n + 1)]
    for j in range(1, n + 1):
        r[j] = p[j]
        for i in range(1, j):
            r[j] = max(r[j], p[i] + r[j - i] - c)
    return r[n]
```

15.1-4

Modify MEMOIZED-CUT-ROD to return not only the value but the actual solution, too.

```
def cut_rod_sub(p, n, r, s):
    if r[n] >= 0:
        return r[n]
    r[n] = 0
    for i in range(1, n + 1):
        ret = p[i] + cut_rod_sub(p, n - i, r, s)
        if r[n] < ret:
            r[n] = ret
            s[n] = i
    return r[n]

def cut_rod(p, n):
    r = [-1 for _ in xrange(n + 1)]
    s = [i for i in xrange(n + 1)]
    cut_rod_sub(p, n, r, s)
    r = r[n]
    subs = []
    while n > 0:
        subs.append(s[n])
        n -= s[n]
    return r, subs
```

15.1-5

The Fibonacci numbers are defined by recurrence (3.22). Give an $O(n)$ -time dynamic-programming algorithm to compute the n th Fibonacci number. Draw the subproblem graph. How many vertices and edges are in the graph?

```
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    a, b = 0, 1
    for i in range(1, n):
        c = a + b
        a, b = b, c
    return c
```

15.2 Matrix-chain multiplication

15.2-1

Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$.

Table m:

	1	2	3	4	5	6
1	0	150	330	405	1655	2010
2		0	360	330	2430	1950
3			0	180	930	1770
4				0	3000	1860
5					0	1500
6						0

Table s:

	1	2	3	4	5	6
1		2	2	2	4	2
2			2	2	2	2
3				3	3	4
4					4	4
5						5
6						

Optimal parenthesization:

$$(A_1 \times A_2) \times ((A_3 \times A_4) \times (A_5 \times A_6))$$

15.2-2

Give a recursive algorithm MATRIX-CHAIN-MULTIPLY (A, s, i, j) that actually performs the optimal matrix-chain multiplication, given the sequence of matrices $\langle A_1, A_2, \dots, A_{n_i} \rangle$, the s table computed by MATRIX-CHAIN-ORDER, and the indices i and j . (The initial call would be MATRIX-CHAIN-MULTIPLY $(A, s, 1, n)$.)

```
MATRIX-CHAIN-MULTIPLY(A, s, i, j)
1 if i == j
2   return A[i]
3 if i + 1 == j
4   return A[i] * A[j]
5 b = MATRIX-CHAIN-MULTIPLY(A, s, i, s[i, j])
6 c = MATRIX-CHAIN-MULTIPLY(A, s, s[i, j] + 1, j)
7 return b * c
```

15.2-3

Use the substitution method to show that the solution to the recurrence (15.6) is $\Omega(2^n)$

Suppose $P(n) \geq c2^n$,

$$\begin{aligned} P(n) &\geq \sum_{k=1}^{n-1} c2^k \cdot c2^{n-k} \\ &= \sum_{k=1}^{n-1} c^2 2^n \\ &= c^2(n-1)2^n \\ &\geq c^2 2^n \quad (n > 1) \\ &\geq c2^n \quad (0 < c \leq 1) \end{aligned}$$

15.2-4

Describe the subproblem graph for matrix-chain multiplication with an input chain of length n . How many vertices does it have? How many edges does it have, and which edges are they?

Vertices: $O(n^2)$, edges: $O(n^3)$.

15.2-5

Let $R(i, j)$ be the number of times that table entry $m[i, j]$ is referenced while computing other table entries in a call of MATRIX-CHAIN-ORDER. Show that the total

$$\sum_{i=1}^n \sum_{j=i}^n R(i, j) = \frac{n^3 - n}{3}$$

number of references for the entire table is

$$\sum_{i=1}^n \sum_{j=i}^n R(i, j) = \sum_{l=2}^n 2(n-l+1)(l-1) = \frac{n^3 - n}{3}$$

15.2-6

Show that a full parenthesization of an n -element expression has exactly $n - 1$ pairs of parentheses.

$n - 1$ multiplications.

15.3 Elements of dynamic programming

15.3-1

Which is a more efficient way to determine the optimal number of multiplications in a matrix-chain multiplication problem: enumerating all the ways of parenthesizing the product and computing the number of multiplications for each, or running RECURSIVE-MATRIX-CHAIN? Justify your answer.

RECURSIVE-MATRIX-CHAIN

15.3-2

Draw the recursion tree for the MERGE-SORT procedure from Section 2.3.1 on an array of 16 elements. Explain why memoization fails to speed up a good divide-and-conquer algorithm such as MERGE-SORT.

It's not overlapping.

15.3-3

Consider a variant of the matrix-chain multiplication problem in which the goal is to parenthesize the sequence of matrices so as to maximize, rather than minimize, the number of scalar multiplications. Does this problem exhibit optimal substructure?

Yes.

15.3-4

As stated, in dynamic programming we first solve the subproblems and then choose which of them to use in an optimal solution to the problem. Professor Capulet claims that we do not always need to solve all the subproblems in order to find an optimal solution. She suggests that we can find an optimal solution to the matrix-chain multiplication problem by always choosing the matrix A_k at which to split the subproduct $A_i A_{i+1} \cdots A_j$ (by selecting k to minimize the quantity $p_{i-1} p_k p_j$) before solving the subproblems. Find an instance of the matrix-chain multiplication problem for which this greedy approach yields a suboptimal solution.

15.3-5

Suppose that in the rod-cutting problem of Section 15.1, we also had limit l_i on the number of pieces of length i that we are allowed to produce, for $i = 1, 2, \dots, n$. Show that the optimal-substructure property described in Section 15.1 no longer holds.

Not independent.

15.3-6

Imagine that you wish to exchange one currency for another. You realize that instead of directly exchanging one currency for another, you might be better off making a series of trades through other currencies, winding up with the currency you want. Suppose that you can trade n different currencies, numbered $1, 2, \dots, n$, where you start with currency 1 and wish to wind up with currency n . You are given, for each pair of currencies i and j , an exchange rate r_{ij} , meaning that if you start with d units of currency i , you can trade for dr_{ij} units of currency j . A sequence of trades may entail a commission, which depends on the number of trades you make. Let c_k be the commission that you are charged when you make k trades. Show that, if $c_k = 0$ for all $k = 1, 2, \dots, n$, then the problem of finding the best sequence of exchanges from currency 1 to currency n exhibits optimal substructure. Then show that if commissions c_k are arbitrary values, then the problem of finding the best sequence of exchanges from currency 1 to currency n does not necessarily exhibit optimal substructure.

$$c_k = 0, r_{ij} = \max_k r_{ik} \cdot r_{kj}.$$

If c_k are arbitrary values, then it's not independent.

15.4 Longest common subsequence

15.4-1

Determine an LCS of $\langle 1, 0, 0, 1, 0, 1, 0, 1 \rangle$ and $\langle 0, 1, 0, 1, 1, 0, 1, 1, 0 \rangle$.
 $\langle 1, 0, 0, 1, 1, 0 \rangle$

15.4-2

Give pseudocode to reconstruct an LCS from the completed c table and the original sequences $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$ in $O(m + n)$ time, without using the b table.

```
PRINT-LCS(c, X, Y, i, j)
1 if c[i][j] == 0
2     return
3 if X[i] == Y[j]
4     PRINT-LCS(c, X, Y, i - 1, j - 1)
5     print X[i]
6 elseif c[i - 1][j] > c[i][j - 1]
7     PRINT-LCS(c, X, Y, i - 1, j)
8 else
9     PRINT-LCS(c, X, Y, i, j - 1)
```

15.4-3

Give a memoized version of LCS-LENGTH that runs in $O(mn)$ time.

```
LCS-LENGTH(X, Y, i, j)
1 if c[i, j] > -1
2     return c[i, j]
3 if i == 0 or j == 0
4     return c[i, j] = 0
5 if xi = yj
6     return c[i, j] = LCS-LENGTH(X, Y, i - 1, j - 1) + 1
7 return c[i, j] = max(LCS-LENGTH(X, Y, i - 1, j), LCS-LENGTH(X, Y, i, j - 1))
```

15.4-4

Show how to compute the length of an LCS using only $2 \cdot \min(m, n)$ entries in the c table plus $O(1)$ additional space. Then show how to do the same thing, but using $\min(m, n)$ entries plus $O(1)$ additional space.

$2 \cdot \min(m, n)$: rolling.

$\min(m, n)$: save the old value of the last computed position.

15.4-5

Give an $O(n^2)$ -time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers.

Calculate the LCS of the original sequence and the sorted sequence,
 $O(n \lg n) + O(n^2) = O(n^2)$ time.

15.4-6 *

Give an $O(n \lg n)$ -time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers.

Binary search.

15.5 Optimal binary search trees

15.5-1

Write pseudocode for the procedure CONSTRUCT-OPTIMAL-BST (*root*) which, given the table root, outputs the structure of an optimal binary search tree. For the example in Figure 15.10, your procedure should print out the structure corresponding to the optimal binary search tree shown in Figure 15.9(b).

```

CONSTRUCT-OPTIMAL-BST(root, i, j, last=0)
1 if i > j
2     return
3 if last == 0
4     print root[i, j] + "is the root"
5 elseif j < last:
6     print root[i, j] + "is the left child of" + last
7 else
8     print root[i, j] + "is the right child of" + last
9 CONSTRUCT-OPTIMAL-BST(root, i, root[i, j] - 1, root[i, j])
10 CONSTRUCT-OPTIMAL-BST(root, root[i, j] + 1, j, root[i, j])

```

15.5-2

Determine the cost and structure of an optimal binary search tree for a set of $n = 7$ keys with the following probabilities

k_4 is the root k_2 is the left child of k_4 k_1 is the left child of k_2 d_0 is the right child of k_1 d_1 is the right child of k_1 k_3 is the right child of k_2 d_2 is the left child of k_3 d_3 is the right child of k_3 k_5 is the right child of k_4 d_4 is the left child of k_5 d_5 is the right child of k_5

15.5-3

Suppose that instead of maintaining the table $w[i, j]$, we computed the value of $w(i, j)$ directly from equation (15.12) in line 9 of OPTIMAL-BST and used this computed value in line 11. How would this change affect the asymptotic running time of OPTIMAL-BST?

$O(n^3)$

15.5-4 *

Knuth [212] has shown that there are always roots of optimal subtrees such that $\text{root}[i, j - 1] \leq \text{root}[i, j] \leq \text{root}[i + 1, j]$ for all $1 \leq i < j \leq n$. Use this fact to modify the OPTIMAL-BST procedure to run in $O(n^2)$ time.

```
10 for r = root[i, j-1] to root[i + 1, j]
```

Problems

15-1 Longest simple path in a directed acyclic graph

Suppose that we are given a directed acyclic graph $G = (V, E)$ with real-valued edge weights and two distinguished vertices s and t . Describe a dynamic-programming approach for finding a longest weighted simple path from s to t . What does the subproblem graph look like? What is the efficiency of your algorithm?

Topological sort.

15-2 Longest palindrome subsequence

A **palindrome** is a nonempty string over some alphabet that reads the same forward and backward. Examples of palindromes are all strings of length 1, civic, racecar, and aibohphobia (fear of palindromes). Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input string. For example, given the input character, your algorithm should return carac. What is the running time of your algorithm?

LCS of the original string and the reversed string.

15-3 Bitonic euclidean traveling-salesman problem

In the **euclidean traveling-salesman problem**, we are given a set of n points in the plane, and we wish to find the shortest closed tour that connects all n points. Figure 15.11(a) shows the solution to a 7-point problem. The general problem is NP-hard, and its solution is therefore believed to require more than polynomial time (see Chapter 34). J. L. Bentley has suggested that we simplify the problem by restricting our attention to **bitonic tours**, that is, tours that start at the leftmost point, go strictly rightward to the rightmost point, and then go strictly leftward back to the starting point. Figure 15.11(b) shows the shortest bitonic tour of the same 7 points. In this case, a polynomial-time algorithm is possible. Describe an $O(n^2)$ -time algorithm for determining an optimal bitonic tour. You may assume that no two points have the same x -coordinate and that all operations on real numbers take unit time.

Sort the points by their x -coordinates, and suppose there are n points. Let $dp[i][j]$ be the minimal distance from i to the first point and from the first point to j . Since $dp[i][j]$ is

symmetric, suppose that $j < i$, then $\min_j dp[j][n]$ is the shortest bitnoic tour. If

$$\begin{aligned} j = i - 1, \quad dp[i][j] &= \min_{k=1}^{i-2} dp[i-1][k] + dist(k, i) && ; \text{ if } j < i - 1, \\ dp[i][j] &= dp[i-1][j] + dist(i-1, i) \end{aligned}$$

15-4 Printing neatly

Consider the problem of neatly printing a paragraph with a monospaced font (all characters having the same width) on a printer. The input text is a sequence of n words of lengths l_1, l_2, \dots, l_n , measured in characters. We want to print this paragraph neatly on a number of lines that hold a maximum of M characters each. Our criterion of "neatness" is as follows. If a given line contains words i through j , where $i \leq j$, and we leave exactly one space between words, the number of extra space characters at the end of the line is $M - j + i - \sum_{k=i}^j l_k$, which must be nonnegative so that the words fit on the line. We wish to minimize the sum, over all lines except the last, of the cubes of the numbers of extra space characters at the ends of lines. Give a dynamic-programming algorithm to print a paragraph of n words neatly on a printer. Analyze the running time and space requirements of your algorithm.

Let $dp[m]$ be the minimal sum when we finished printing m words.

$$dp[m] = dp[m - k] + cost(m - k + 1, m)$$

$$cost(i, j) = \begin{cases} \infty & \text{if } M - j + i - \sum_{k=i}^j l_k < 0 \\ 0 & \text{if } j = n \\ \left(M - j + i - \sum_{k=i}^j l_k \right)^3 & \text{otherwise} \end{cases}$$

15-5 Edit distance

In order to transform one source string of text $x[1 \dots m]$ to a target string $y[1 \dots n]$, we can perform various transformation operations. Our goal is, given x and y , to produce a series of transformations that change x to y . We use an array z —assumed to be large enough to hold all the characters it will need—to hold the intermediate results. Initially, z is empty, and at termination, we should have $z[j] = y[j]$ for $j = 1, 2, \dots, n$. We maintain current indices i into x and j into z , and the operations are allowed to alter z and these indices. Initially, $i = j = 1$. We are required to examine every character in x during the transformation, which means that at the end of the sequence of transformation operations, we must have $i = m + 1$.

We may choose from among six transformation operations:

Copy a character from x to z by setting $z[j] = x[i]$ and then incrementing both i and j . This operation examines $x[i]$.

Replace a character from x by another character c , by setting $z[j] = c$, and then incrementing both i and j . This operation examines $x[i]$.

Delete a character from x by incrementing i but leaving j alone. This operation examines $x[i]$.

Insert the character c into z by setting $z[j] = c$ and then incrementing j , but leaving i alone. This operation examines no characters of x .

Twiddle (i.e., exchange) the next two characters by copying them from x to z but in the opposite order; we do so by setting $z[j] = x[i + 1]$ and $z[j + 1] = x[i]$ and then setting $i = i + 2$ and $j = j + 2$. This operation examines $x[i]$ and $x[i + 1]$.

Kill the remainder of x by setting $i = m + 1$. This operation examines all characters in x that have not yet been examined. This operation, if performed, must be the final operation.

- a. Given two sequences $x[1 \dots m]$ and $y[1 \dots n]$ and set of transformation-operation costs, the **edit distance** from x to y is the cost of the least expensive operatoin sequence that transforms x to y . Describe a dynamic-programming algorithm that finds the edit distance from $x[1 \dots m]$ to $y[1 \dots n]$ and prints an optimal opeartion sequence. Analyze the running time and space requirements of your algorithm.

- **Copy:** $dp[i][j] = dp[i - 1][j - 1] + cost(copy)$ if $x[i] = y[j]$
- **Replace:** $dp[i][j] = dp[i - 1][j - 1] + cost(replace)$ if $x[i] \neq y[j]$
- **Delete:** $dp[i][j] = dp[i - 1][j] + cost(delete)$
- **Insert:** $dp[i][j] = dp[i][j - 1] + cost(insert)$
- **Twiddle:** $dp[i][j] = dp[i - 2][j - 2] + cost(twiddle)$ if $x[i - 1] = y[j]$ and $x[i] = y[j - 1]$
- **Kill:** $dp[i][j] = \min_{k=1}^i dp[k][j] + cost(kill)$ if $j = n$

The edit-distance problem generalizes the problem of aligning two DNA sequences (see, for example, Setubal and Meidanis [310, Section 3.2]). There are several methods for measuring the similarity of two DNA sequences by aligning them. One such method to align two sequences x and y consists of inserting spaces at arbitrary locations in the two sequences (including at either end) so that the resulting sequences x' and y' have the same length but do not have a space in the same position (i.e., for no position j are both $x'[j]$ and $y'[j]$ a space). Then we assign a "score" to each position. Position j receives a score as follows:

- $+1$ if $x'[j] = y'[j]$ and neither is a space,
- -1 if $x'[j] \neq y'[j]$ and neither is a space,
- -2 if either $x'[j]$ or $y'[j]$ is a space.

b. Explain how to cast the problem of finding an optimal alignment as an edit distance problem using a subset of the transformation operations copy, replace, delete, insert, twiddle, and kill.

$$cost(copy) = +1, cost(replace) = -1, cost(delete) = cost(insert) = 1$$

15-6 Planning a company party

Professor Stewart is consulting for the president of a corporation that is planning a company party. The company has a hierarchical structure; that is, the supervisor relation forms a tree rooted at the president. The personnel office has ranked each employee with a conviviality rating, which is a real number. In order to make the party fun for all attendees, the president does not want both an employee and his or her immediate supervisor to attend.

Professor Stewart is given the tree that describes the structure of the corporation, using the left-child, right-sibling representation described in Section 10.4. Each node of the tree holds, in addition to the pointers, the name of an employee and that employee's conviviality ranking. Describe an algorithm to make up a guest list that maximizes the sum of the conviviality ratings of the guests. Analyze the running time of your algorithm.

Let $dp[i][j]$ be the maximal sum rooted at i , $j = 0$ means i will not attend, $j = 1$ means i will attend. $dp[i][0] = \max_j (dp[j][0], dp[j][1])$, $dp[i][1] = \max_j dp[j][0]$.

15-7 Viterbi algorithm

We can use dynamic programming on a directed graph $G = (V, E)$ for speech recognition. Each edge $(u, v) \in E$ is labeled with a sound $\sigma(u, v)$ from a finite set Σ of sounds. The labeled graph is a formal model of a person speaking a restricted language. Each path in the graph starting from a distinguished vertex $v_0 \in V$ corresponds to a possible sequence of sounds produced by the model. We define the label of a directed path to be the concatenation of the labels of the edges on that path.

a. Describe an efficient algorithm that, given an edge-labeled graph G with distinguished vertex v_0 and a sequence $s = \langle \sigma_1, \sigma_2, \dots, \sigma_k \rangle$ of sounds from Σ , returns a path in G that begins at v_0 and has s as its label, if any such path exists. Otherwise, the algorithm should return NO-SUCH-PATH. Analyze the running time of your algorithm.

Let $dp[i][j]$ be the state of vertex j in iteration i , $dp[0][v_0] = \text{true}$.

$$dp[i][j] = \begin{cases} 1 & \exists_k dp[i-1][k] = 1 \text{ and } \sigma(k, i) = \sigma_i \\ 0 & \text{otherwise} \end{cases}$$

Now, suppose that every edge $(u, v) \in E$ has an associated nonnegative probability $p(u, v)$ of traversing the edge (u, v) from vertex u and thus producing the corresponding sound. The sum of the probabilities of the edges leaving any vertex equals 1. The probability of a path is defined to be the product of the probabilities of its edges. We can view the probability of a path beginning at v_0 as the probability that a "random walk" beginning at v_0 will follow the specified path, where we randomly choose which edge to take leaving a vertex u according to the probabilities of the available edges leaving u .

- b.** Extend your answer to part (a) so that if a path is returned, it is a *most probable path* starting at v_0 and having label s . Analyze the running time of your algorithm.

$$dp[0][v_0] = 1.0$$

$$dp[i][j] = \max_{dp[i-1][k]=1 \text{ and } \sigma(k,i)=\sigma_i} dp[i-1][k] \cdot p(k, i)$$

15-8 Image compression by seam carving

We are given a color picture consisting of an $m \times n$ array $A[1 \dots m, 1 \dots n]$ of pixels, where each pixel specifies a triple of red, green, and blue (RGB) intensities. Suppose that we wish to compress this picture slightly. Specifically, we wish to remove one pixel from each of the m rows, so that the whole picture becomes one pixel narrower. To avoid disturbing visual effects, however, we require that the pixels removed in two adjacent rows be in the same or adjacent columns; the pixels removed form a "seam" from the top row to the bottom row where successive pixels in the seam are adjacent vertically or diagonally.

- a.** Show that the number of such possible seams grows at least exponentially in m , assuming that $n > 1$.

$$\text{num} \geq 2^n$$

- b.** Suppose now that along with each pixel $A[i, j]$, we have calculated a real-valued disruption measure $d[i, j]$, indicating how disruptive it would be to remove pixel $A[i, j]$. Intuitively, the lower a pixel's disruption measure, the more similar the pixel is to its neighbors. Suppose further that we define the disruption measure of a seam to be the sum of the disruption measures of its pixels.

Give an algorithm to find a seam with the lowest disruption measure. How efficient is your algorithm?

$$dp[i, j] = d[i, j] + \min(dp[i - 1, j - 1], dp[i - 1, j], dp[i - 1, j + 1])$$

15-9 Breaking a string

A certain string-processing language allows a programmer to break a string into two pieces. Because this operation copies the string, it costs n time units to break a string of n characters into two pieces. Suppose a programmer wants to break a string into many pieces. The order in which the breaks occur can affect the total amount of time used. For example, suppose that the programmer wants to break a 20-character string after characters 2, 8, and 10 (numbering the characters in ascending order from the left-hand end, starting from 1). If she programs the breaks to occur in left-to-right order, then the first break costs 20 time units, the second break costs 18 time units (breaking the string from characters 3 to 20 at character 8), and the third break costs 12 time units, totaling 50 time units. If she programs the breaks to occur in right-to-left order, however, then the first break costs 20 time units, the second break costs 10 time units, and the third break costs 8 time units, totaling 38 time units. In yet another order, she could break first at 8 (costing 20), then break the left piece at 2 (costing 8), and finally the right piece at 10 (costing 12), for a total cost of 40.

Design an algorithm that, given the numbers of characters after which to break, determines a least-cost way to sequence those breaks. More formally, given a string S with n characters and an array $L[1 \dots m]$ containing the break points, compute the lowest cost for a sequence of breaks, along with a sequence of breaks that achieves this cost.

$$dp[i][j] = \text{len}(i, j) + \min_{k=i+1}^{j-1} (dp[i][k] + dp[k+1][j])$$

15-10 Planning an investment strategy

Your knowledge of algorithms helps you obtain an exciting job with the Acme Computer Company, along with a \$10,000 signing bonus. You decide to invest this money with the goal of maximizing your return at the end of 10 years. You decide to use the Amalgamated Investment Company to manage your investments. Amalgamated Investments requires you to observe the following rules. It offers n different investments, numbered 1 through n . In each year j , investment i provides a return rate of r_{ij} . In other words, if you invest d dollars in investment i in year j , then at the end of year j , you have dr_{ij} dollars. The return rates are guaranteed, that is, you are given all the return rates for the next 10 years for each investment. You make investment decisions only once per year. At the end of each year, you can leave the money made in the previous year in the same investments, or you can shift money to other investments, by either shifting money between existing investments or moving money to a new investment. If you do not move your money between two consecutive years, you pay a fee of f_1 dollars, whereas if you switch your money, you pay a fee of f_2 dollars, where $f_2 > f_1$.

- a. The problem, as stated, allows you to invest your money in multiple investments in each year. Prove that there exists an optimal investment strategy that, in each year, puts all the money into a single investment. (Recall that an optimal investment strategy maximizes the amount of money after 10 years and is not concerned with any other objectives, such as minimizing risk.)
- b. Prove that the problem of planning your optimal investment strategy exhibits optimal substructure.
- c. Design an algorithm that plans your optimal investment strategy. What is the running time of your algorithm? Let $dp[j][i]$ be the maximal profit in year j with the last investment i .

$$dp[j][i] = \begin{cases} dp[j-1][k] \cdot r_{kj} + f_2 & (k \neq i) \\ dp[j-1][i] \cdot r_{ij} + f_1 & \end{cases}$$

- d. Suppose that Amalgamated Investments imposed the additional restriction that, at any point, you can have no more than \$15,000 in any one investment. Show that the problem of maximizing your income at the end of 10 years no longer exhibits optimal substructure.

15-11 Inventory planning

The Rinky Dink Company makes machines that resurface ice rinks. The demand for such products varies from month to month, and so the company needs to develop a strategy to plan its manufacturing given the fluctuating, but predictable, demand. The company wishes to design a plan for the next n months. For each month i , the company knows the demand d_i , that is, the number of machines that it will sell. Let $D = \sum_{i=1}^n d_i$ be the total demand over the next n months. The company keeps a full-time staff who provide labor to manufacture up to m machines per month. If the company needs to make more than m machines in a given month, it can hire additional, part-time labor, at a cost that works out to c dollars per machine. Furthermore, if, at the end of a month, the company is holding any unsold machines, it must pay inventory costs. The cost for holding j machines is given as a function $h(j)$ for $j = 1, 2, \dots, D$, where $h(j) \geq 0$ for $1 \leq j \leq D$ and $h(j) \leq h(j+1)$ for $1 \leq j \leq D-1$.

Give an algorithm that calculates a plan for the company that minimizes its costs while fulfilling all the demand. The running time should be polynomial in n and D .

Let $dp[i][j]$ be the minimal cost after month i with j machines remained.

$$dp[i][0] = \begin{cases} \min_j dp[i-1][j] & (j \leq d_i \text{ and } j+m \geq d_i) \\ \min_j dp[i-1][j] + c \cdot (d_i - j - m) & (j+m \leq d_i) \end{cases}$$

$$dp[i][j] = \min_k dp[i-1][k] + h(k+m-d_i)$$

15-12 Signing free-agent baseball players

Suppose that you are the general manager for a major-league baseball team. During the off-season, you need to sign some free-agent players for your team. The team owner has given you a budget of $\$X$ to spend on free agents. You are allowed to spend less than $\$X$ altogether, but the owner will fire you if you spend any more than $\$X$.

You are considering N different positions, and for each position, P free-agent players who play that position are available. Because you do not want to overload your roster with too many players at any position, for each position you may sign at most one free agent who plays that position. (If you do not sign any players at a particular position, then you plan to stick with the players you already have at that position.)

To determine how valuable a player is going to be, you decide to use a sabermetric statistic⁹ known as "VORP", or "value over replacement player". A player with a higher VORP is more valuable than a player with a lower VORP. A player with a higher VORP is not necessarily more expensive to sign than a player with a lower VORP, because factors other than a player's value determine how much it costs to sign him.

For each available free-agent player, you have three pieces of information:

- the player's position,
- the amount of money it will cost to sign the player, and
- the player's VORP.

Devise an algorithm that maximizes the total VORP of the players you sign while spending no more than $\$X$ altogether. You may assume that each player signs for a multiple of \$100,000. Your algorithm should output the total VORP of the players you sign, the total amount of money you spend, and a list of which players you sign.

Analyze the running time and space requirement of your algorithm.

16 Greedy Algorithm

- 16.1 An activity-selection problem
- 16.2 Elements of the greedy strategy
- 16.3 Huffman codes
- 16.4 Matroids and greedy methods
- 16.5 A task-scheduling problem as a matroid
- Problems

16.1 An activity-selection problem

16.1-1

Give a dynamic-programming algorithm for the activity-selection problem, based on recurrence (16.2). Have your algorithm compute the sizes $c[i, j]$ as defined above and also produce the maximum-size subset of mutually compatible activities.

Assume that the inputs have been sorted as in equation (16.1). Compare the running time of your solution to the running time of GREEDY-ACTIVITY-SELECTOR.

$O(n^3)$

16.1-2

Suppose that instead of always selecting the first activity to finish, we instead select the last activity to start that is compatible with all previously selected activities. Describe how this approach is a greedy algorithm, and prove that it yields an optimal solution.

The same.

16.1-3

Not just any greedy approach to the activity-selection problem produces a maximum-size set of mutually compatible activities. Give an example to show that the approach of selecting the activity of least duration from among those that are compatible with previously selected activities does not work. Do the same for the approaches of always selecting the compatible activity that overlaps the fewest other remaining activities and always selecting the compatible remaining activity with the earliest start time.

Least duration: [1, 5], [4, 7], [6, 10]

Overlap fewest: [1, 4], [5, 7], [8, 10], [1, 2], [3, 5], [6, 8], [9, 10], ...

Earliest start: [1, 6], [5, 10], [2, 4]

16.1-4

Suppose that we have a set of activities to schedule among a large number of lecture halls, where any activity can take place in any lecture hall. We wish to schedule all the activities using as few lecture halls as possible. Give an efficient greedy algorithm to determine which activity should use which lecture hall.

(This problem is also known as the *interval-graph coloring problem*. We can create an interval graph whose vertices are the given activities and whose edges connect incompatible activities. The smallest number of colors required to color every vertex so that no two adjacent vertices have the same color corresponds to finding the fewest lecture halls needed to schedule all of the given activities.)

Sort the intervals by start time, if the start time of one interval is the same as the finish time of the other interval, we should assume the finish time is less than the start time. From left to right, add 1 when there is a start time and subtract 1 when there is a finish time, the number of halls needed is the maximum number of the count.

16.1-5

Consider a modification to the activity-selection problem in which each activity a_i has, in addition to a start and finish time, a value v_i . The objective is no longer to maximize the number of activities scheduled, but instead to maximize the total value of the activities scheduled. That is, we wish to choose a set A of compatible activities such that $\sum_{a_k \in A} v_k$ is maximized. Give a polynomial-time algorithm for this problem.

Let $dp[i]$ be the maximum total value before time i ,

$$dp[i] = \max(dp[i - 1], \max_{f_j \leq i} dp[s_j] + v_j)$$

```
def activity_selection(s, f, v):
    dp = {}
    n = len(s)
    last = None
    for i in sorted(list(set(s + f))):
        if last is None:
            dp[i] = 0
        else:
            dp[i] = last
            for j in range(n):
                if f[j] <= i:
                    dp[i] = max(dp[i], dp[s[j]] + v[j])
        last = dp[i]
    return last
```

$\Theta(n^2)$

16.2 Elements of the greedy strategy

16.2-1

Prove that the fractional knapsack problem has the greedy-choice property.

Obviously

16.2-2

Give a dynamic-programming solution to the 0-1 knapsack problem that runs in $O(nW)$ time, where n is the number of items and W is the maximum weight of items that the thief can put in his knapsack.

```
def zero_one_knapsack(v, w, W):
    n = len(v)
    dp = [0] * (W + 1)
    for i in range(n):
        for j in range(w, w[i] - 1, -1):
            dp[j] = max(dp[j], dp[j - w[i]] + v[i])
    return dp[W]
```

16.2-3

Suppose that in a 0-1 knapsack problem, the order of the items when sorted by increasing weight is the same as their order when sorted by decreasing value. Give an efficient algorithm to find an optimal solution to this variant of the knapsack problem, and argue that your algorithm is correct.

Suppose in an optimal solution we take an item with v_1, w_1 , and drop an item with v_2, w_2 , and $w_1 > w_2, v_1 < v_2$, we can substitute 1 with 2 and get a better solution.

Therefore we should always choose the items with the greatest values.

16.2-4

Professor Gekko has always dreamed of inline skating across North Dakota. He plans to cross the state on highway U.S. 2, which runs from Grand Forks, on the eastern border with Minnesota, to Williston, near the western border with Montana. The professor can carry two liters of water, and he can skate m miles before running out of water. (Because North Dakota is relatively flat, the professor does not have to worry about drinking water at a greater rate on uphill sections than on flat or downhill sections.) The professor will start in Grand Forks with two full liters of water. His official North Dakota state map shows all the places along U.S. 2 at which he can refill his water and the distances between these locations.

The professor's goal is to minimize the number of water stops along his route across the state. Give an efficient method by which he can determine which water stops he should make. Prove that your strategy yields an optimal solution, and give its running time.

Go to the furthest stop within m miles in each iteration.

16.2-5

Describe an efficient algorithm that, given a set $\{x_1, x_2, \dots, x_n\}$ of points on the real line, determines the smallest set of unit-length closed intervals that contains all of the given points. Argue that your algorithm is correct.

Place the left side of the unit-interval to the first left-most uncovered point in each iteration.

16.2-6 *

Show how to solve the fractional knapsack problem in $O(n)$ time.

Choose the median of v_i/w_i in $O(n)$, partition the sequence with the median in $O(n)$, if the sum of weights in the more valuable side is less or equal to W , we take all the items in this side and repeat the steps in the other side; otherwise we repeat the steps in the more valuable side. The algorithm runs in $T(n) = T(n/2) + O(n)$, which is $O(n)$.

16.2-7

Suppose you are given two sets A and B , each containing n positive integers. You can choose to reorder each set however you like. After reordering, let a_i be the i th element of set A , and let b_i be the i th element of set B . You then receive a payoff of $\prod_{i=1}^n a_i^{b_i}$. Give an algorithm that will maximize your payoff. Prove that your algorithm maximizes the payoff, and state its running time.

Sort A and B into monotonically increasing/decreasing order.

16.3 Huffman codes

16.3-1

Explain why, in the proof of Lemma 16.2, if $x.freq = b.freq$, then we must have $a.freq = b.freq = x.freq = y.freq$.

16.3-2

Prove that a binary tree that is not full cannot correspond to an optimal prefix code.

16.3-3

What is an optimal Huffman code for the following set of frequencies, based on the first 8 Fibonacci numbers?

a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21

Can you generalize your answer to find the optimal code when the frequencies are the first n Fibonacci numbers?

- a: 1111111
- b: 1111110
- c: 111110
- d: 11110
- e: 1110
- f: 110
- g: 10
- h: 0

16.3-4

Prove that we can also express the total cost of a tree for a code as the sum, over all internal nodes, of the combined frequencies of the two children of the node.

16.3-5

Prove that if we order the characters in an alphabet so that their frequencies are monotonically decreasing, then there exists an optimal code whose codeword lengths are monotonically increasing.

16.3-6

Suppose we have an optimal prefix code on a set $C = \{0, 1, \dots, n - 1\}$ of characters and we wish to transmit this code using as few bits as possible. Show how to represent any optimal prefix code on C using only $2n - 1 + n \lceil \lg n \rceil$ bits.

Use one bit for representing internal or leaf node, which is $2n - 1$ bits.

16.3-7

Generalize Huffman's algorithm to ternary codewords (i.e., codewords using the symbols 0, 1, and 2), and prove that it yields optimal ternary codes.

Merge three nodes.

16.3-8

Suppose that a data file contains a sequence of 8-bit characters such that all 256 characters are about equally common: the maximum character frequency is less than twice the minimum character frequency. Prove that Huffman coding in this case is no more efficient than using an ordinary 8-bit fixed-length code.

Full binary tree, another 8-bit encoding.

16.3-9

Show that no compression scheme can expect to compress a file of randomly chosen 8-bit characters by even a single bit.

$$2^n >> 2^{n-1}$$

16.4 Matroids and greedy methods

16.4-1

Show that (S, I_k) is a matroid, where S is any finite set and I_k is the set of all subsets of S of size at most k , where $k \leq |S|$.

16.4-2 *

Given an $m \times n$ matrix T over some field (such as the reals), show that (S, I) is a matroid, where S is the set of columns of T and $A \in I$ if and only if the columns in A are linearly independent.

16.4-3 *

Show that if (S, I) is a matroid, then (S, I') is a matroid, where

$$I' = \{A' : S - A' \text{ contains some maximal } A \in I\}.$$

That is, the maximal independent sets of (S, I') are just the complements of the maximal independent sets of (S, I) .

16.4-4 *

Let S be a finite set and let S_1, S_2, \dots, S_k be a partition of S into nonempty disjoint subsets. Define the structure (S, I) by the condition that $I = \{A : |A \cap S_i| \leq 1 \text{ for } i = 1, 2, \dots, k\}$. Show that (S, I) is a matroid. That is, the set of all sets A that contain at most one member of each subset in the partition determines the independent sets of a matroid.

16.4-5

Show how to transform the weight function of a weighted matroid problem, where the desired optimal solution is a *minimum-weight* maximal independent subset, to make it a standard weighted-matroid problem. Argue carefully that your transformation is correct.

16.5 A task-scheduling problem as a matroid

16.5-1

Solve the instance of the scheduling problem given in Figure 16.7, but with each penalty w_i replaced by $80 - w_i$.

a_i	1	2	3	4	5	6	7
d_i	4	2	4	3	1	4	6
w_i	10	20	30	40	50	60	70

$\langle a_5, a_4, a_3, a_6, a_7 \rangle$, $w_1 + w_2 = 30$.

16.5-2

Show how to use property 2 of Lemma 16.12 to determine in time $O(|A|)$ whether or not a given set A of tasks is independent.

Inserting by deadline.

Problems

16-1 Coin changing

Consider the problem of making change for n cents using the fewest number of coins. Assume that each coin's value is an integer.

a. Describe a greedy algorithm to make change consisting of quarters, dimes, nickels, and pennies. Prove that your algorithm yields an optimal solution.

Use the coin as large as possible.

b. Suppose that the available coins are in the denominations that are powers of c , i.e., the denominations are c^0, c^1, \dots, c^k for some integers $c > 1$ and $k \geq 1$. Show that the greedy algorithm always yields an optimal solution.

Same.

c. Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Your set should include a penny so that there is a solution for every value of n .

$\langle 10, 9, 1 \rangle$

For 18, the greedy algorithm yields 9 coins, the optimal solution is $\langle 9, 9 \rangle$, which contains 2 coins.

d. Give an $O(nk)$ -time algorithm that makes change for any set of k different coin denominations, assuming that one of the coins is a penny.

Let $dp[i]$ be the minimal number of coins of amount i , $dp[i] = 1 + \min_j dp[i - c_j]$.

16-2 Scheduling to minimize average completion time

Suppose you are given a set $S = \{a_1, a_2, \dots, a_n\}$ of tasks, where task a_i requires p_i units of processing time to complete, once it has started. You have one computer on which to run these tasks, and the computer can run only one task at a time. Let c_i be the **completion time** of task a_i , that is, the time at which task a_i completes processing. Your goal is to minimize the average completion time, that is, to minimize $(1/n) \sum_{i=1}^n c_i$. For example, suppose there are two tasks, a_1 and a_2 , with $p_1 = 3$ and $p_2 = 5$, and consider the schedule in which a_2 runs first, followed by a_1 . Then $c_2 = 5$, $c_1 = 8$, and the average completion time is $(5 + 8)/2 = 6.5$. If task a_1 runs first, however, then $c_1 = 3$, $c_2 = 8$, and the average completion time is $(3 + 8)/2 = 5.5$.

- a.** Give an algorithm that schedules the tasks so as to minimize the average completion time. Each task must run non-preemptively, that is, once task a_i starts, it must run continuously for p_i units of time. Prove that your algorithm minimizes the average completion time, and state the running time of your algorithm.

Suppose a permutation of S is $\langle r_1, r_2, \dots, r_n \rangle$, the total completion time is

$$\sum_{i=1}^n (n - i + 1) \cdot p_{r_i}$$

. The optimal solution is to sort p_i into increasing order.

- b.** Suppose now that the tasks are not all available at once. That is, each task cannot start until its **release time** r_i . Suppose also that we allow **preemption**, so that a task can be suspended and restarted at a later time. For example, a task a_i with processing time $p_i = 6$ and release time $r_i = 1$ might start running at time 1 and be preempted at time 4. It might then resume at time 10 but be preempted at time 11, and it might finally resume at time 13 and complete at time 15. Task a_i has run for a total of 6 time units, but its running time has been divided into three pieces. In this scenario, a_i 's completion time is 15. Give an algorithm that schedules the tasks so as to minimize the average completion time in this new scenario. Prove that your algorithm minimizes the average completion time, and state the running time of your algorithm.

Preemption will not yield a better solution if there is no new task. Each time there is a new task, assume that the current running task is preempted, let the current condition be a new scheduling task without preemption.

16-3 Acyclic subgraphs

a. The **incidence matrix** for an undirected graph $G = (V, E)$ is a $|V| \times |E|$ matrix M such that $M_{ve} = 1$ if edge e is incident on vertex v , and $M_{ve} = 0$ otherwise.

Argue that a set of columns of M is linearly independent over the field of integers modulo 2 if and only if the corresponding set of edges is acyclic. Then, use the result of Exercise 16.4-2 to provide an alternate proof that (E, I) of part (a) is a matroid.

b. Suppose that we associate a nonnegative weight $w(e)$ with each edge in an undirected graph $G = (V, E)$. Give an efficient algorithm to find an acyclic subset of E of maximum total weight.

Maximum spanning tree.

c. Let $G(V, E)$ be an arbitrary directed graph, and let (E, I) be defined so that $A \in I$ if and only if A does not contain any directed cycles. Give an example of a directed graph G such that the associated system (E, I) is not a matroid. Specify which defining condition for a matroid fails to hold.

d. The **incidence matrix** for a directed graph $G = (V, E)$ with no self-loops is a $|V| \times |E|$ matrix M such that $M_{ve} = -1$ if edge e leaves vertex v , $M_{ve} = 1$ if edge e enters vertex v , and $M_{ve} = 0$ otherwise. Argue that if a set of columns of M is linearly independent, then the corresponding set of edges does not contain a directed cycle.

e. Exercise 16.4-2 tells us that the set of linearly independent sets of columns of any matrix M forms a matroid. Explain carefully why the results of parts (d) and (e) are not contradictory. How can there fail to be a perfect correspondence between the notion of a set of edges being acyclic and the notion of the associated set of columns of the incidence matrix being linearly independent?

16-4 Scheduling variations

Consider the following algorithm for the problem from Section 16.5 of scheduling unit-time tasks with deadlines and penalties. Let all n time slots be initially empty, where time slot i is the unit-length slot of time that finishes at time i . We consider the tasks in order of monotonically decreasing penalty. When considering task a_j , if there exists a time slot at or before a_j 's deadline d_j that is still empty, assign a_j to the latest such slot, filling it. If there is no such slot, assign task a_j to the latest of the as yet unfilled slots.

- a.** Argue that this algorithm always gives an optimal answer.
- b.** Use the fast disjoint-set forest presented in Section 21.3 to implement the algorithm efficiently. Assume that the set of input tasks has already been sorted into monotonically decreasing order by penalty. Analyze the running time of your implementation.

16-5 Off-line caching

- a.** Write pseudocode for a cache manager that uses the furthest-in-future strategy. The input should be a sequence $\langle r_1, r_2, \dots, r_n \rangle$ of requests and a cache size k , and the output should be a sequence of decisions about which data element (if any) to evict upon each request. What is the running time of your algorithm?
- b.** Show that the off-line caching problem exhibits optimal substructure.
- c.** Prove that furthest-in-future produces the minimum possible number of cache misses.

17 Amortized Analysis

- 17.1 Aggregate analysis
- 17.2 The accounting method
- 17.3 The potential method
- 17.4 Dynamic tables
- Problems 1
- Problems 2

17.1 Aggregate analysis

17.1-1

If the set of stack operations included a MULTIPUSH operation, which pushes k items onto the stack, would the $O(1)$ bound on the amortized cost of stack operations continue to hold?

No.

17.1-2

Show that if a DECREMENT operation were included in the k -bit counter example, n operations could cost as much as $\Theta(nk)$ time.

Increment and decrement repeatedly on $011 \dots 11$.

17.1-3

Suppose we perform a sequence of n operations on a data structure in which the i th operation costs i if i is an exact power of 2, and 1 otherwise. Use aggregate analysis to determine the amortized cost per operation.

$$1 + 2 + 2^2 + \dots + 2^{\lfloor \lg n \rfloor} \leq 2n$$

17.2 The accounting method

17.2-1

Suppose we perform a sequence of stack operations on a stack whose size never exceeds k . After every k operations, we make a copy of the entire stack for backup purposes. Show that the cost of n stack operations, including copying the stack, is $O(n)$ by assigning suitable amortized costs to the various stack operations.

PUSH: 2 , POP: 2 , COPY: 0 .

17.2-2

Redo Exercise 17.1-3 using an accounting method of analysis.

Insert: 3 .

17.2-3

Suppose we wish not only to increment a counter but also to reset it to zero (i.e., make all bits in it 0). Counting the time to examine or modify a bit as $\Theta(1)$, show how to implement a counter as an array of bits so that any sequence of n INCREMENT and RESET operations takes time $O(n)$ on an initially zero counter.

Twice cost of each bit.

17.3 The potential method

17.3-1

Suppose we have a potential function Φ such that $\Phi(D_i) \geq \Phi(D_0)$ for all i , but $\Phi(D_0) \neq 0$. Show that there exists a potential function Φ' such that $\Phi'(D_0) = 0$, $\Phi'(D_i) \geq 0$ for all $i \geq 1$, and the amortized costs using Φ' are the same as the amortized costs using Φ .

$$\Phi'(D_i) = \Phi(D_i) - \Phi(D_0)$$

17.3-2

Redo Exercise 17.1-3 using a potential method of analysis.

$$\Phi(D_i) = 2 \cdot 2^{\lceil \lg i \rceil} - i$$

17.3-3

Consider an ordinary binary min-heap data structure with n elements supporting the instructions INSERT and EXTRACT-MIN in $O(\lg n)$ worst-case time. Give a potential function Φ such that the amortized cost of INSERT is $O(\lg n)$ and the amortized cost of EXTRACT-MIN is $O(1)$, and show that it works.

$$\Phi(D_i) = \text{number of elements in the heap} \times \lg n$$

$$\text{INSERT: } \Phi(D_i) = O(\lg n) + \lg n = O(\lg n)$$

$$\text{EXTRACT-MIN: } \Phi(D_i) = O(\lg n) - \lg n = O(1)$$

17.3-4

What is the total cost of executing n of the stack operations PUSH, POP, and MULTIPOP, assuming that the stack begins with s_0 objects and finishes with s_n objects?

$$\begin{aligned}\sum_{i=1}^n c_i &= \sum_{i=1}^n \hat{c}_i - \Phi(D_n) + \Phi(D_0) \\ &= n - s_n + s_0\end{aligned}$$

17.3-5

Suppose that a counter begins at a number with b 1s in its binary representation, rather than at 0. Show that the cost of performing n INCREMENT operations is $O(n)$ if $n = \Omega(b)$. (Do not assume that b is constant.)

$$\begin{aligned}\sum_{i=1}^n c_i &= \sum_{i=1}^n \hat{c}_i - \Phi(D_n) + \Phi(D_0) \\ &= n - x + b \\ &\leq n - x + n \\ &= O(n)\end{aligned}$$

17.3-6

Show how to implement a queue with two ordinary stacks (Exercise 10.1-6) so that the amortized cost of each ENQUEUE and each DEQUEUE operation is $O(1)$.

$\Phi(D_i)$ = number of elements in the first stack.

17.3-7

Design a data structure to support the following two operations for a dynamic multiset S of integers, which allows duplicate values:

INSERT (S, x) inserts x into S .

DELETE-LARGER-HALF (S) deletes the largest $\lceil |S|/2 \rceil$ elements from S .

Explain how to implement this data structure so that any sequence of m INSERT and DELETE-LARGER-HALF operations runs in $O(m)$ time. Your implementation should also include a way to output the elements of S in $O(|S|)$ time.

An array of elements.

INSERT: push the element to the back of the array.

DELETE-LARGER-HALF: find the median in $O(n)$ and delete the first $\lceil |S|/2 \rceil$ elements that are larger or equal to the median.

17.4 Dynamic tables

17.4-1

Suppose that we wish to implement a dynamic, open-address hash table. Why might we consider the table to be full when its load factor reaches some value α that is strictly less than 1? Describe briefly how to make insertion into a dynamic, open-address hash table run in such a way that the expected value of the amortized cost per insertion is $O(1)$. Why is the expected value of the actual cost per insertion not necessarily $O(1)$ for all insertions?

17.4-2

Show that if $\alpha_{i-1} \geq 1/2$ and the i th operation on a dynamic table is TABLE-DELETE, then the amortized cost of the operation with respect to the potential function (17.6) is bounded above by a constant.

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot \text{num}_i - \text{size}_i) - (2 \cdot (\text{num}_i + 1) - \text{size}_i) \\ &= -1\end{aligned}$$

17.4-3

Suppose that instead of contracting a table by halving its size when its load factor drops below $1/4$, we contract it by multiplying its size by $2/3$ when its load factor drops below $1/3$. Using the potential function

$$\Phi(T) = |2 \cdot T.\text{num} - T.\text{size}|,$$

show that the amortized cost of a TABLE-DELETE that uses this strategy is bounded above by a constant.

If $1/3 < \alpha_i \leq 1/2$,

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (\text{size}_i - 2 \cdot \text{num}_i) - (\text{size}_i - 2 \cdot (\text{num}_i + 1)) \\ &= 3\end{aligned}$$

If the i th operation does trigger a contraction,

$$\begin{aligned}\frac{1}{3}size_{i-1} &= num_i + 1 \\ size_{i-1} &= 3(num_i + 1) \\ size_i &= \frac{2}{3}size_{i-1} = 2(num_i + 1)\end{aligned}$$

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= (num_i + 1) + [2 \cdot (num_i + 1) - 2 \cdot num_i] - [3 \cdot (num_i + 1) - 2 \cdot (num_i + 1)] \\ &= 2\end{aligned}$$

Problems

17-1 Bit-reversed binary counter

Chapter 30 examines an important algorithm called the fast Fourier transform, or FFT. The first step of the FFT algorithm performs a ***bit-reversal permutation*** on an input array $A[0 \dots n - 1]$ whose length is $n = 2^k$ for some nonnegative integer k . This permutation swaps elements whose indices have binary representations that are the reverse of each other.

We can express each index a as a k -bit sequence $\langle a_{k-1}, a_{k-2}, \dots, a_0 \rangle$, where $a = \sum_{i=0}^{k-1} a_i 2^i$. We define

$$\text{rev}_k(\langle a_{k-1}, a_{k-2}, \dots, a_0 \rangle) = \langle a_0, a_1, \dots, a_{k-1} \rangle,$$

thus,

$$\text{rev}_k(a) = \sum_{i=0}^{k-1} a_{k-i-1} 2^i$$

For example, if $n = 16$ (or, equivalently, $k = 4$), then $\text{rev}_k(3) = 12$, since the 4-bit representation of 3 is 0011, which when reversed gives 1100, the 4-bit representation of 12.

- a. Given a function rev_k that runs in $\Theta(k)$ time, write an algorithm to perform the bit-reversal permutation on an array of length $n = 2^k$ in $O(nk)$ time.

```
def rev_k(k, a):
    x = 0
    for _ in xrange(k):
        x <= 1
        x += a & 1
        a >= 1
    return x
```

We can use an algorithm based on an amortized analysis to improve the running time of the bit-reversal permutation. We maintain a "bit-reversed counter" and a procedure BIT-REVERSED-INCREMENT that, when given a bit-reversed-counter value a , produces $\text{rev}_k(\text{rev}_k(a) + 1)$. If $k = 4$, for example, and the bit-reversed counter starts at 0, then successive calls to BIT-REVERSED-INCREMENT produce the sequence

$$0000, 1000, 0100, 1100, 0010, 1010, \dots = 0, 8, 4, 12, 2, 10, \dots$$

b. Assume that the words in your computer store k -bit values and that in unit time, your computer can manipulate the binary values with operations such as shifting left or right by arbitrary amounts, bitwise-AND, bitwise-OR, etc. Describe an implementation of the BIT-REVERSED-INCREMENT procedure that allows the bit-reversal permutation on an n -element array to be performed in a total of $O(n)$ time.

```
class BitReversedCounter:
    def __init__(self, k):
        self.k = k
        self.c = 0

    def increment(self):
        for i in xrange(self.k - 1, -1, -1):
            self.c ^= 1 << i
            if self.c & (1 << i) > 0:
                break
        return self.c
```

c. Suppose that you can shift a word left or right by only one bit in unit time. Is it still possible to implement an $O(n)$ -time bit-reversal permutation?

```
class BitReversedCounter:
    def __init__(self, k):
        self.k = k
        self.c = 0
        self.n = 1 << (self.k - 1)

    def increment(self):
        i = self.n
        for _ in xrange(self.k - 1, -1, -1):
            self.c ^= i
            if self.c & i > 0:
                break
            i >>= 1
        return self.c
```

17-2 Making binary search dynamic

Binary search of a sorted array takes logarithmic search time, but the time to insert a new element is linear in the size of the array. We can improve the time for insertion by keeping several sorted arrays.

Specifically, suppose that we wish to support SEARCH and INSERT on a set of n elements. Let $k = \lceil \lg(n + 1) \rceil$, and let the binary representation of n be $\langle n_{k-1}, n_{k-2}, \dots, n_0 \rangle$. We have k sorted arrays A_0, A_1, \dots, A_{k-1} , where for $i = 0, 1, \dots, k - 1$, the length of array A_i is 2^i . Each array is either full or empty, depending on whether $n_i = 1$ or $n_i = 0$, respectively. The total number of elements held in all k arrays is therefore $\sum_{i=0}^{k-1} n_i 2^i = n$. Although each individual array is sorted, elements in different arrays bear no particular relationship to each other.

a. Describe how to perform the SEARCH operation for this data structure. Analyze its worst-case running time.

$O(\lg^2 n)$

b. Describe how to perform the INSERT operation. Analyze its worst-case and amortized running times.

Merge sort.

Worst: $O(n)$

Amortized: $O(\lg n)$

c. Discuss how to implement DELETE.

17-3 Amortized weight-balanced trees

Consider an ordinary binary search tree augmented by adding to each node x the attribute $x.size$ giving the number of keys stored in the subtree rooted at x . Let α be a constant in the range $1/2 \leq \alpha < 1$. We say that a given node x is α -**balanced** if $x.left.size \leq \alpha \cdot x.size$ and $x.right.size \leq \alpha \cdot x.size$. The tree as a whole is α -**balanced** if every node in the tree is α -balanced. The following amortized approach to maintaining weight-balanced trees was suggested by G. Varghese.

- a.** A $1/2$ -balanced tree is, in a sense, as balanced as it can be. Given a node x in an arbitrary binary search tree, show how to rebuild the subtree rooted at x so that it becomes $1/2$ -balanced. Your algorithm should run in time $\Theta(x.size)$, and it can use $O(x.size)$ auxiliary storage.

Choose the middle node as the root.

- b.** Show that performing a search in an n -node α -balanced binary search tree takes $O(\lg n)$ worst-case time.

Let $\beta = 1/\alpha$, $\beta^k = n$, $k = \log_\beta n = O(\log n) = O(\lg n)$.

For the remainder of this problem, assume that the constant α is strictly greater than $1/2$. Suppose that we implement INSERT and DELETE as usual for an n -node binary search tree, except that after every such operation, if any node in the tree is no longer α -balanced, then we "rebuild" the subtree rooted at the highest such node in the tree so that it becomes $1/2$ -balanced.

We shall analyze this rebuilding scheme using the potential method. For a node x in a binary search tree T , we define

$$\Delta(x) = |x.left.size - x.right.size|,$$

and we define the potential of T as

$$\Phi(T) = c \sum_{x \in T: \Delta(x) \geq 2} \Delta(x),$$

where c is a sufficiently large constant that depends on α .

- c.** Argue that any binary search tree has nonnegative potential and that a $1/2$ -balanced tree has potential 0.

$\Delta(x) \geq 0$: nonnegative potential.

$1/2$ -balanced: $\Delta(x) \leq 1$, $\Phi(T) = 0$.

d. Suppose that m units of potential can pay for rebuilding an m -node subtree. How large must c be in terms of α in order for it to take $O(1)$ amortized time to rebuild a subtree that is not α -balanced?

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ O(1) &= m + \Phi(D_i) - \Phi(D_{i-1}) \\ \Phi(D_{i-1}) &= m + \Phi(D_i) \\ \Phi(D_{i-1}) &\geq m \\ \\ \Delta(x) &= x.\text{left.size} - x.\text{right.size} \\ &\geq \alpha \cdot m - ((1 - \alpha)m - 1) \\ &= (2\alpha - 1)m + 1 \\ \\ m &\leq c((2\alpha - 1)m + 1) \\ c &\geq \frac{m}{(2\alpha - 1)m + 1} \\ &\geq \frac{1}{2\alpha}\end{aligned}$$

e. Show that inserting a node into or deleting a node from an n -node α -balanced tree costs $O(\lg n)$ amortized time.

17-4 The cost of restructuring red-black trees

There are four basic operations on red-black trees that perform *structural modifications*: node insertions, node deletions, rotations, and color changes. We have seen that RB-INSERT and RB-DELETE use only $O(1)$ rotations, node insertions, and node deletions to maintain the red-black properties, but they may make many more color changes.

- a. Describe a legal red-black tree with n nodes such that calling RB-INSERT to add the $(n + 1)^{\text{st}}$ node causes $\Omega(\lg n)$ color changes. Then describe a legal red-black tree with n nodes for which calling RB-DELETE on a particular node causes $\Omega(\lg n)$ color changes.

Insert: a complete red-black tree in which all nodes have different color with their parents.

Delete: a complete red-black tree in which all nodes are black.

Although the worst-case number of color changes per operation can be logarithmic, we shall prove that any sequence of m RB-INSERT and RB-DELETE operations on an initially empty red-black tree causes $O(m)$ structural modifications in the worst case. Note that we count each color change as a structural modification.

- b. Some of the cases handled by the main loop of the code of both RB-INSERT-FIXUP and RB-DELETE-FIXUP are terminating: once encountered, they cause the loop to terminate after a constant number of additional operations. For each of the cases of RB-INSERT-FIXUP and RB-DELETE-FIXUP, specify which are terminating and which are not.

RB-INSERT-FIXUP: all cases except for case 1.

RB-DELETE-FIXUP: case 2.

We shall first analyze the structural modifications when only insertions are performed.

Let T be a red-black tree, and define $\Phi(T)$ to be the number of red nodes in T .

Assume that 1 unit of potential can pay for the structural modifications performed by any of the three cases of RB-INSERT-FIXUP.

- c. Let T' be the result of applying Case 1 of RB-INSERT-FIXUP to T . Argue that $\Phi(T') = \Phi(T) - 1$.

Parent and uncle: red to black.

Grandparent: black to red.

d. When we insert a node into a red-black tree using RB-INSERT, we can break the operation into three parts. List the structural modifications and potential changes resulting from lines 1–16 of RB-INSERT, from nonterminating cases of RB-INSERT-FIXUP, and from terminating cases of RB-INSERT-FIXUP.

Case 1: decrease by 1.

Case 2 & 3: no effect.

e. Using part (d), argue that the amortized number of structural modifications performed by any call of RB-INSERT is $O(1)$.

$O(1)$

We now wish to prove that there are $O(m)$ structural modifications when there are both insertions and deletions. Let us define, for each node x ,

$$w(x) = \begin{cases} 0 & \text{if } x \text{ is red,} \\ 1 & \text{if } x \text{ is black and has no red children,} \\ 0 & \text{if } x \text{ is black and has one red children,} \\ 2 & \text{if } x \text{ is black and has two red children,} \end{cases}$$

Now we redefine the potential of a red-black tree T as

$$\Phi(T) = \sum_{x \in T} w(x)$$

and let T' be the tree that results from applying any nonterminating case of RB-INSERT-FIXUP or RB-DELETE-FIXUP to T .

f. Show that $\Phi(T') \leq \Phi(T) - 1$ for all nonterminating cases of RB-INSERT-FIXUP.

Argue that the amortized number of structural modifications performed by any call of RB-INSERT-FIXUP is $O(1)$.

$O(1)$

g. Show that $\Phi(T') \leq \Phi(T) - 1$ for all nonterminating cases of RB-DELETE-FIXUP. Argue that the amortized number of structural modifications performed by any call of RB-DELETE-FIXUP is $O(1)$.

$O(1)$

h. Complete the proof that in the worst case, any sequence of m RB-INSERT and RB-DELETE operations performs $O(m)$ structural modifications.

$O(m)$

17-5 Competitive analysis of self-organizing lists with move-to-front

A **self-organizing** list is a linked list of n elements, in which each element has a unique key. When we search for an element in the list, we are given a key, and we want to find an element with that key.

A self-organizing list has two important properties:

1. To find an element in the list, given its key, we must traverse the list from the beginning until we encounter the element with the given key. If that element is the k th element from the start of the list, then the cost to find the element is k .
2. We may reorder the list elements after any operation, according to a given rule with a given cost. We may choose any heuristic we like to decide how to reorder the list.

Assume that we start with a given list of n elements, and we are given an access sequence $\sigma = \langle \sigma_1, \sigma_2, \dots, \sigma_m \rangle$ of keys to find, in order. The cost of the sequence is the sum of the costs of the individual accesses in the sequence.

Out of the various possible ways to reorder the list after an operation, this problem focuses on transposing adjacent list elements—switching their positions in the list—with a unit cost for each transpose operation. You will show, by means of a potential function, that a particular heuristic for reordering the list, move-to-front, entails a total cost no worse than 4 times that of any other heuristic for maintaining the list order—even if the other heuristic knows the access sequence in advance! We call this type of analysis a **competitive analysis**.

For a heuristic H and a given initial ordering of the list, denote the access cost of sequence σ by $C_H(\sigma)$. Let m be the number of accesses in σ .

a. Argue that if heuristic H does not know the access sequence in advance, then the worst-case cost for H on an access sequence σ is $C_H(\sigma) = \Omega(mn)$.

Always last.

With the **move-to-front** heuristic, immediately after searching for an element x , we move x to the first position on the list (i.e., the front of the list).

Let $\text{rank}_L(x)$ denote the rank of element x in list L , that is, the position of x in list L . For example, if x is the fourth element in L , then $\text{rank}_L(x) = 4$. Let c_i denote the cost of access σ_i using the move-to-front heuristic, which includes the cost of finding the element in the list and the cost of moving it to the front of the list by a series of transpositions of adjacent list elements.

b. Show that if σ_i accesses element x in list L using the move-to-front heuristic, then $c_i = 2 \cdot \text{rank}_L(x) - 1$.

Access: $\text{rank}_L(x)$

Move: $\text{rank}_L(x) - 1$

Now we compare move-to-front with any other heuristic H that processes an access sequence according to the two properties above. Heuristic H may transpose elements in the list in any way it wants, and it might even know the entire access sequence in advance.

Let L_i be the list after access σ_i using move-to-front, and let L_{i-1}^* be the list after access σ_i using heuristic H . We denote the cost of access σ_i by c_i for move-to-front and by c_i^* for heuristic H . Suppose that heuristic H performs t_i^* transpositions during access σ_i .

c. In part (b), you showed that $c_i = 2 \cdot \text{rank}_{L_{i-1}}(x) - 1$. Now show that $c_i^* = \text{rank}_{L_{i-1}^*}(x) + t_i^*$.

Access: $\text{rank}_{L_{i-1}^*}(x)$

Move: t_i^*

We define an **inversion** in list L_i as a pair of elements y and z such that y precedes z in L_i and z precedes y in list L_i^* . Suppose that list L_i has q_i inversions after processing the access sequence $\langle \sigma_1, \sigma_2, \dots, \sigma_i \rangle$. Then, we define a potential function Φ that maps L_i to a real number by $\Phi(L_i) = 2q_i$. For example, if L_i has the elements $\langle e, c, a, d, b \rangle$ and L_i^* has the elements $\langle c, a, b, d, e \rangle$, then L_i has 5 inversions $((e, c), (e, a), (e, d), (e, b), (d, b))$, and so $\Phi(L_i) = 10$. Observe that $\Phi(L_i) \geq 0$ for all i and that, if move-to-front and heuristic H start with the same list L_0 , then $\Phi(L_0) = 0$.

- d.** Argue that a transposition either increases the potential by 2 or decreases the potential by 2.

Same before: decrease by 2.

Same after: increase by 2.

Suppose that access σ_i finds the element x . To understand how the potential changes due to σ_i , let us partition the elements other than x into four sets, depending on where they are in the lists just before the i th access:

- Set A consists of elements that precede x in both L_{i-1} and L_{i-1}^* .
- Set B consists of elements that precede x in L_{i-1} and follow x in L_{i-1}^* .
- Set C consists of elements that follow x in L_{i-1} and precede x in L_{i-1}^* .
- Set D consists of elements that follow x in both L_{i-1} and L_{i-1}^* .

- e.** Argue that $\text{rank}_{L_{i-1}}(x) = |A| + |B| + 1$ and $\text{rank}_{L_{i-1}^*}(x) = |A| + |C| + 1$.

Precede.

- f.** Show that access σ_i causes a change in potential of

$$\Phi(L_i) - \Phi(L_{i-1}) \leq 2(|A| - |B| + t_i^*) ,$$

where, as before, heuristic H performs t_i^* transpositions during access σ_i .

Define the amortized cost \hat{c}_i of access σ_i by $\hat{c}_i = c_i + \Phi(L_i) - \Phi(L_{i-1})$.

- g.** Show that the amortized cost \hat{c}_i of access σ_i is bounded from above by $4c_i^*$.

$$\begin{aligned}\hat{c}_i &\leq 2(|A| + |B| + 1) - 1 + 2(|A| - |B| + t_i^*) \\&= 4|A| + 1 + 2t_i^* \\&\leq 4(|A| + |C| + 1 + t_i^*) \\&= 4c_i^*\end{aligned}$$

h. Conclude that the cost $C_{MTF}(\sigma)$ of access sequence σ with move-to-front is at most 4 times the cost $C_H(\sigma)$ of σ with any other heuristic H , assuming that both heuristics start with the same list.

18 B-Trees

- 18.1 Definition of B-trees
- 18.2 Basic operations on B-trees
- 18.3 Deleting a key from a B-tree
- Problems

18.1 Definition of B-trees

18.1-1

Why don't we allow a minimum degree of $t = 1$?

No key.

18.1-2

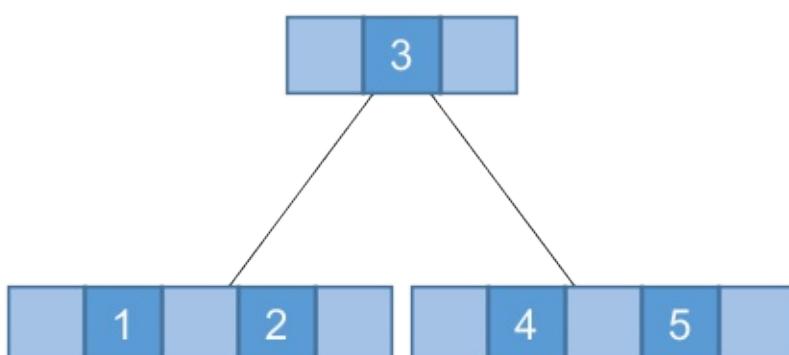
For what values of t is the tree of Figure 18.1 a legal B-tree?

2 or 3.

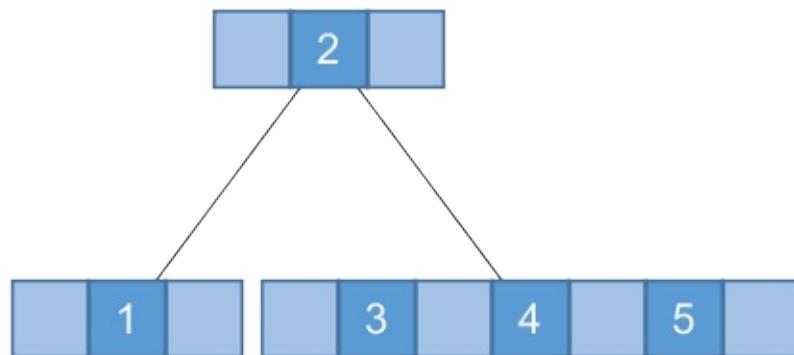
18.1-3

Show all legal B-trees of minimum degree 2 that represent $\{1, 2, 3, 4, 5\}$.

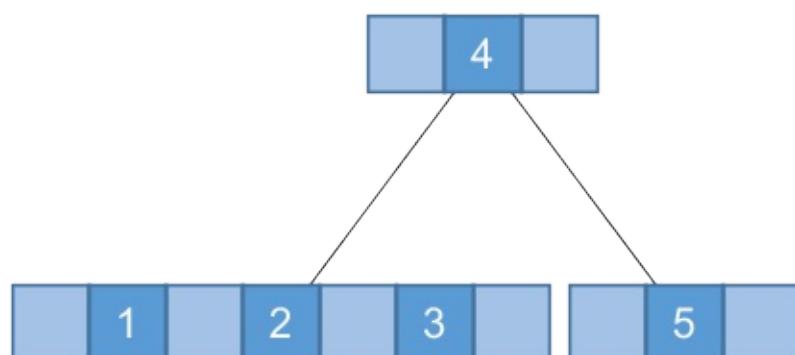
<https://github.com/CyberZH/B>



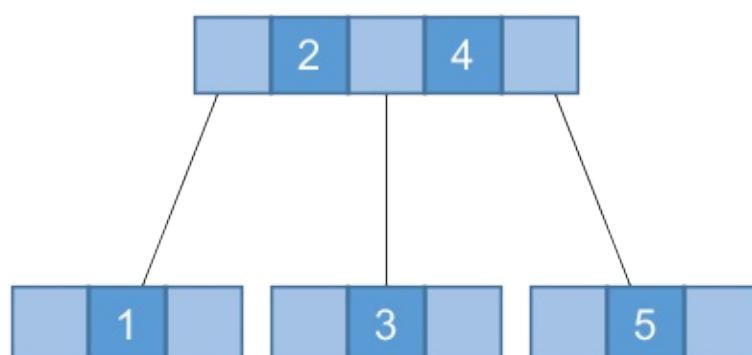
<https://github.com/CyberZH/B>



<https://github.com/CyberZH/B>



<https://github.com/CyberZH/B>



18.1-4

As a function of the minimum degree t , what is the maximum number of keys that can be stored in a B-tree of height h ?

$$\begin{aligned} n &= (1 + 2t + (2t)^2 + \cdots + (2t)^h) \cdot (2t - 1) \\ &= (2t)^{h+1} - 1 \end{aligned}$$

18.1-5

Describe the data structure that would result if each black node in a red-black tree were to absorb its red children, incorporating their children with its own.

$t = 2$, 2-3-4 tree

18.2 Basic operations on B-trees

18.2-1

Show the results of inserting the keys

$F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E$

in order into an empty B-tree with minimum degree 2. Draw only the configurations of the tree just before some node must split, and also draw the final configuration.

Initial state: empty B-tree



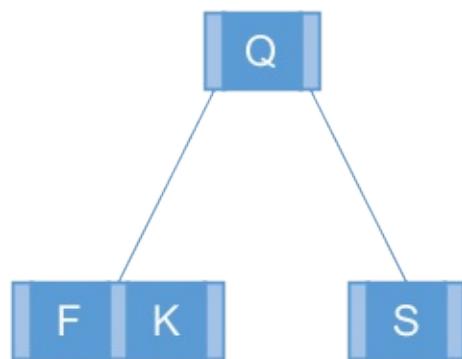
After inserting F: node with one key



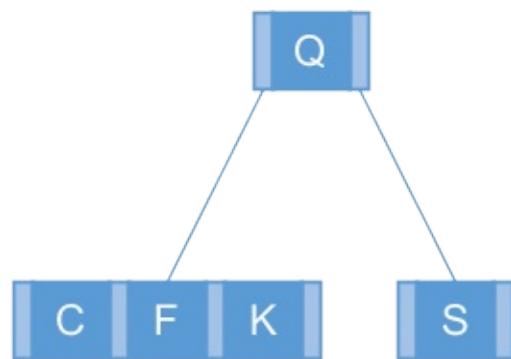
After inserting F and S: node with two keys



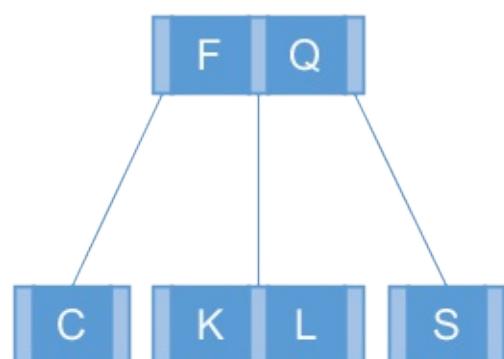
Insertion in a B-tree of order 2



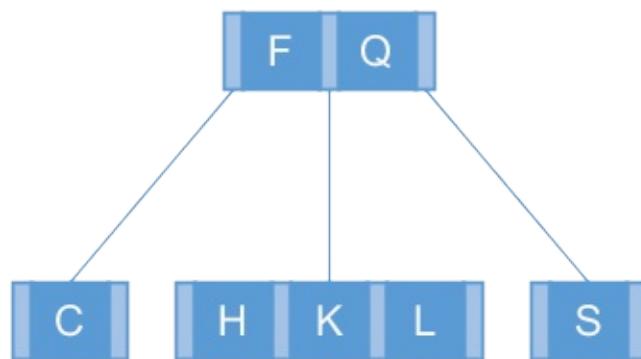
Insertion of key C



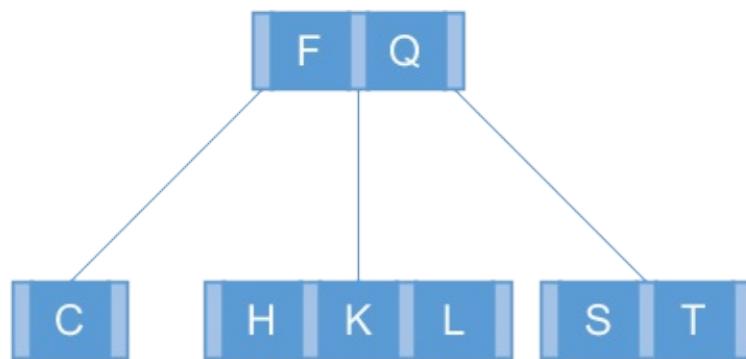
Insertion of key L



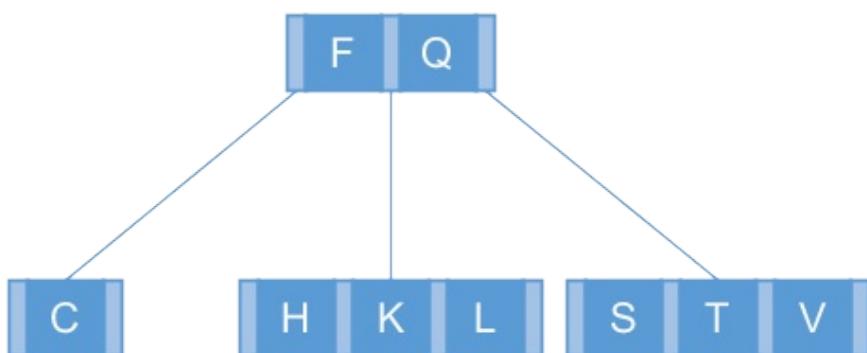
Insertion in a B-tree of order 3



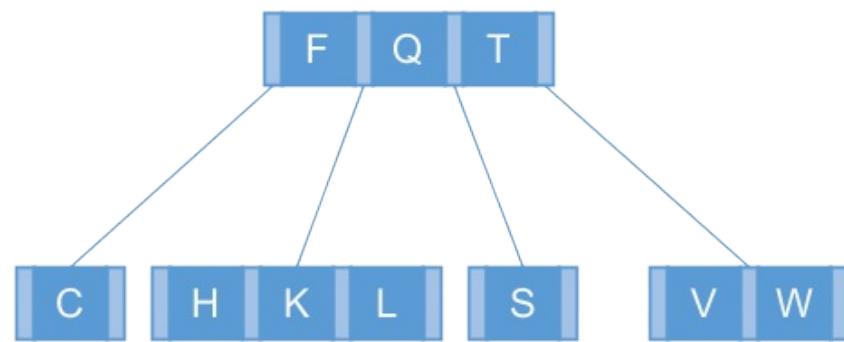
Insertion of 'S' in the B-tree



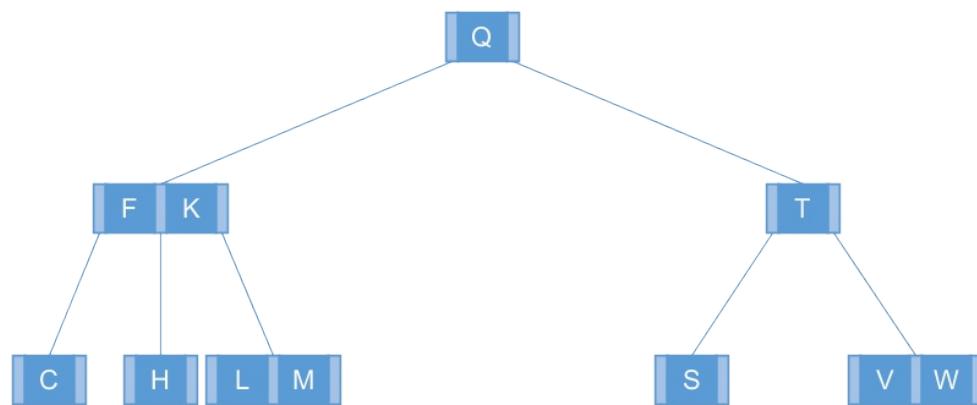
Insertion of 'V' in the B-tree



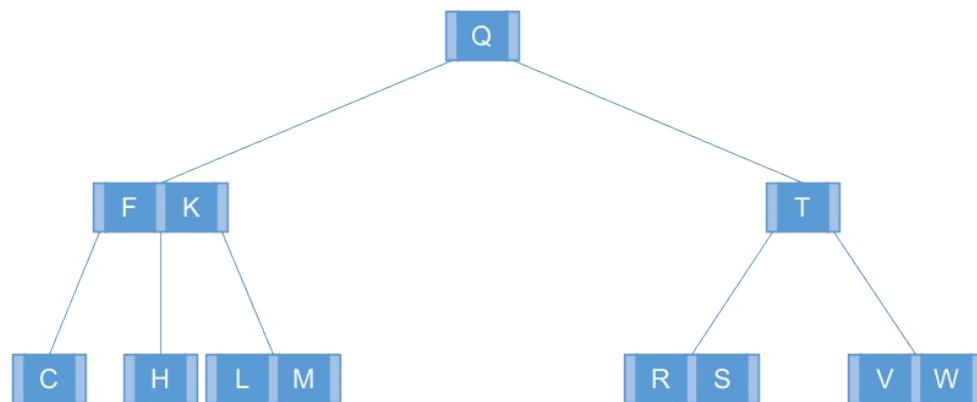
Insertion in a B-tree of order 3



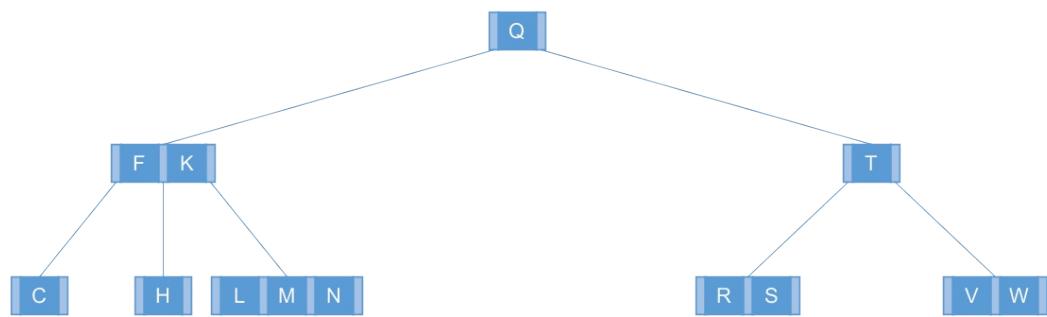
Insertion of key R



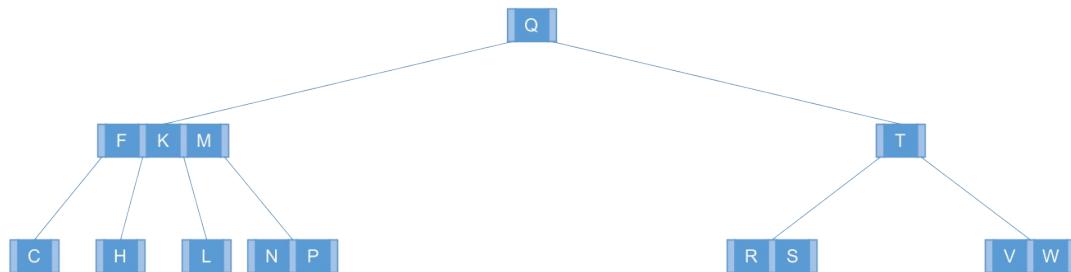
Insertion of key R



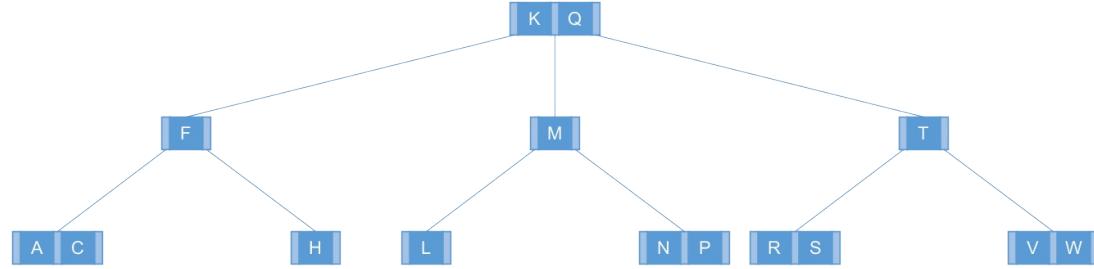
Insertion operation



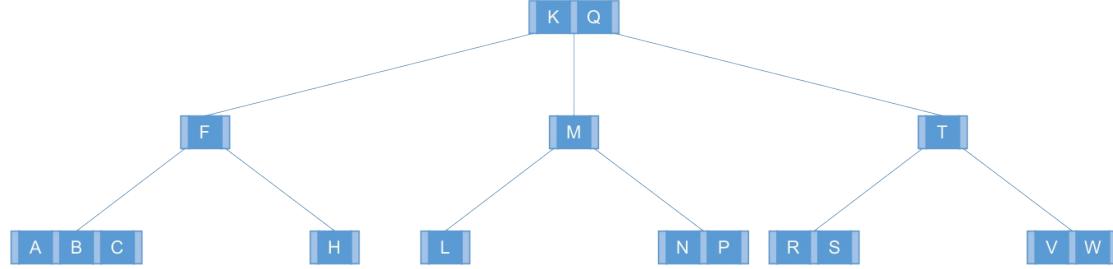
Intermediate state



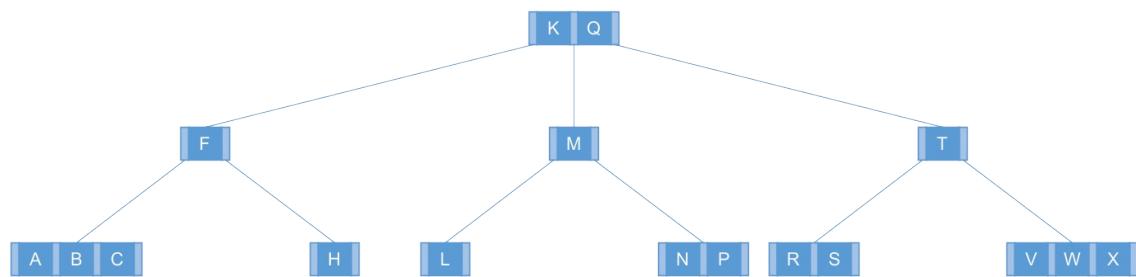
Final state



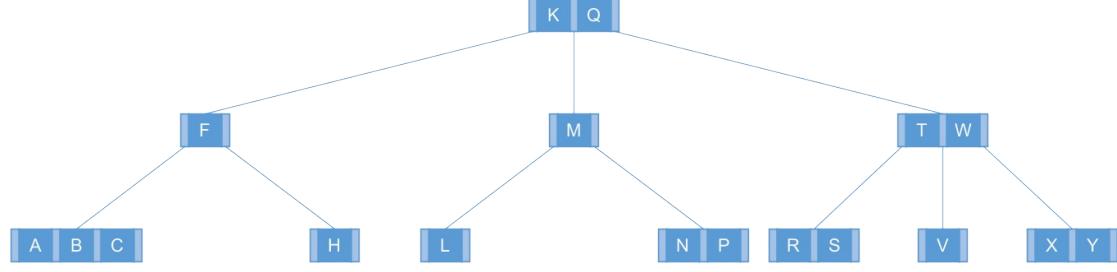
Final state (another view)



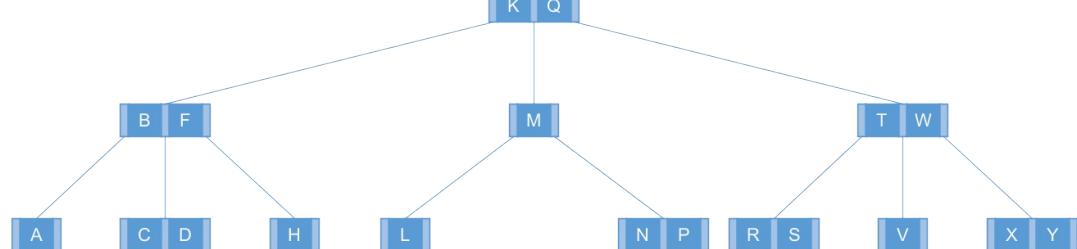
Initial state of the B-tree



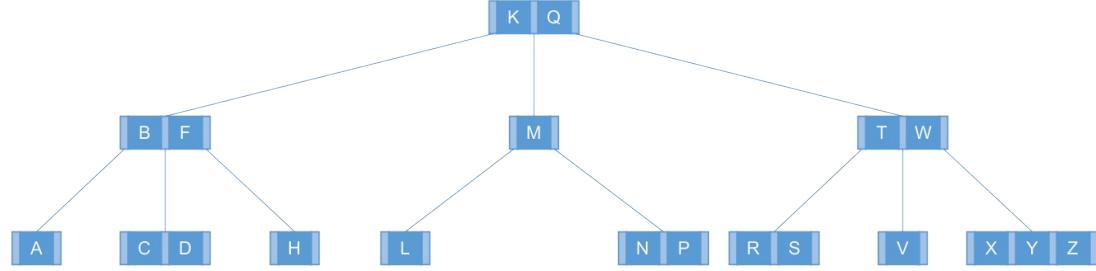
After inserting E



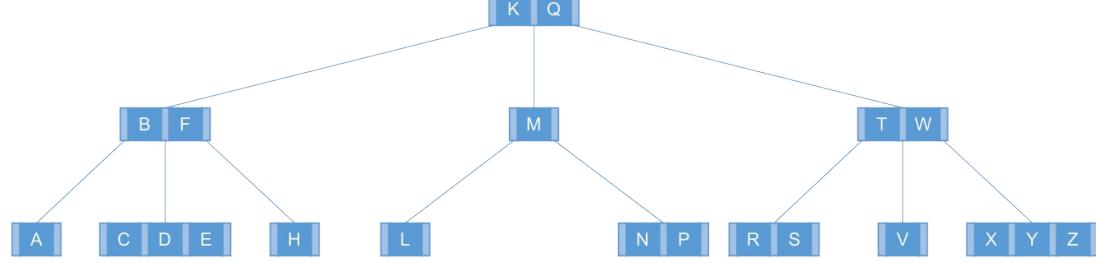
After inserting E



After inserting E



After inserting E



18.2-2

Explain under what circumstances, if any, redundant DISK-READ or DISK-WRITE operations occur during the course of executing a call to B-TREE-INSERT. (A redundant DISK-READ is a DISK-READ for a page that is already in memory. A redundant DISK-WRITE writes to disk a page of information that is identical to what is already stored there.)

No redundant.

18.2-3

Explain how to find the minimum key stored in a B-tree and how to find the predecessor of a given key stored in a B-tree.

```

class BTreeNode:
    def __init__(self, t):
        self.n = 0
        self.key = [None] * (2 * t - 1)
        self.c = [None] * (2 * t)
        self.leaf = True

class BTree:
    def __init__(self, degree):
        self.t = degree
        self.root = BTreeNode(degree)

    def disk_read(self, x):
        pass

    def disk_write(self, x):
        pass

    def split_child(self, x, i):
        t = self.t
        z = BTreeNode(t)
        y = x.c[i]
        z.leaf = y.leaf
        z.n = t - 1
        for j in range(t - 1):
            z.key[j] = y.key[j + t]
        if not y.leaf:
            for j in range(t):
                z.c[j] = y.c[j + t]
        y.n = t - 1
        for j in range(x.n, i - 1, -1):
            x.c[j + 1] = x.c[j]
        x.c[i + 1] = z

```

```

        for j in range(x.n - 1, i - 2, -1):
            x.key[j + 1] = x.key[j]
        x.key[i] = y.key[t - 1]
        x.n += 1
        self.disk_write(y)
        self.disk_write(z)
        self.disk_write(x)

def insert(self, k):
    t = self.t
    r = self.root
    if r.n == 2 * t - 1:
        s = BTreenode(t)
        self.root = s
        s.leaf = False
        s.n = 0
        s.c[0] = r
        self.split_child(s, 0)
        self.insert_nonfull(s, k)
    else:
        self.insert_nonfull(r, k)

def insert_nonfull(self, x, k):
    t = self.t
    i = x.n - 1
    if x.leaf:
        while i >= 0 and k < x.key[i]:
            x.key[i + 1] = x.key[i]
            i -= 1
        x.key[i + 1] = k
        x.n += 1
        self.disk_write(x)
    else:
        while i >= 0 and k < x.key[i]:
            i -= 1
        i += 1
        self.disk_read(x.c[i])
        if x.c[i].n == 2 * t - 1:
            self.split_child(x, i)
            if k > x.key[i]:
                i += 1
        self.insert_nonfull(x.c[i], k)

def minimum(self):
    def minimum_sub(x):
        if x is None:
            return None
        if x.n > 0 and x.c[0] is not None:
            return minimum_sub(x.c[0])
        return x.key[0]
    if self.root.n == 0:
        return None
    return minimum_sub(self.root)

```

```

def predecessor(self, k):
    def predecessor_sub(x):
        if x is None:
            return None
        for i in xrange(x.n - 1, -1, -1):
            if k > x.key[i]:
                c = predecessor_sub(x.c[i + 1])
                if c is None:
                    return x.key[i]
                return max(c, x.key[i])
        return predecessor_sub(x.c[0])
    if self.root.n == 0:
        return None
    return predecessor_sub(self.root)

def successor(self, k):
    def successor_sub(x):
        if x is None:
            return None
        for i in xrange(x.n):
            if k < x.key[i]:
                c = successor_sub(x.c[i])
                if c is None:
                    return x.key[i]
                return min(c, x.key[i])
        return successor_sub(x.c[x.n])
    if self.root.n == 0:
        return None
    return successor_sub(self.root)

```

18.2-4 *

Suppose that we insert the keys $\{1, 2, \dots, n\}$ into an empty B-tree with minimum degree 2. How many nodes does the final B-tree have?

At least $n - 2 \lg(n + 1)$

18.2-5

Since leaf nodes require no pointers to children, they could conceivably use a different (larger) t value than internal nodes for the same disk page size. Show how to modify the procedures for creating and inserting into a B-tree to handle this variation.

18.2-6

Suppose that we were to implement B-TREE-SEARCH to use binary search rather than linear search within each node. Show that this change makes the CPU time required $O(\lg n)$, independently of how t might be chosen as a function of n .

$$\log_t n \cdot \lg t = \lg n$$

18.2-7

Suppose that disk hardware allows us to choose the size of a disk page arbitrarily, but that the time it takes to read the disk page is $a + bt$, where a and b are specified constants and t is the minimum degree for a B-tree using pages of the selected size. Describe how to choose t so as to minimize (approximately) the B-tree search time. Suggest an optimal value of t for the case in which $a = 5$ milliseconds and $b = 10$ microseconds.

$$\min \log_t n \cdot (a + bt) = \min \frac{a + bt}{\ln t}$$

$$\frac{\partial}{\partial t} \left(\frac{a + bt}{\ln t} \right) = -\frac{a + bt - bt \ln t}{t \ln^2 t}$$

$$a + bt = bt \ln t$$

$$5 + 10t = 10t \ln t$$

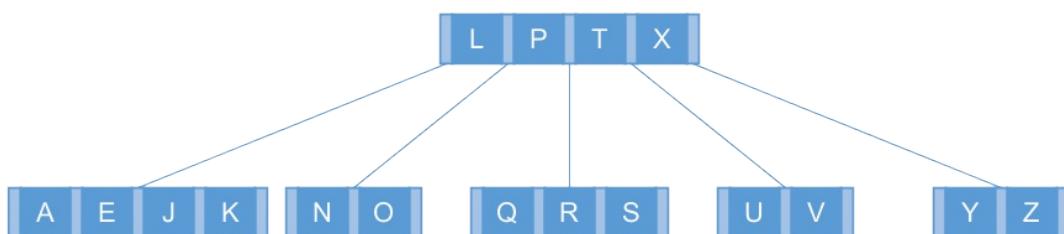
$$t = e^{W\left(\frac{1}{2e}\right)+1}$$

where W is the LambertW function, and we should choose $t = 3$.

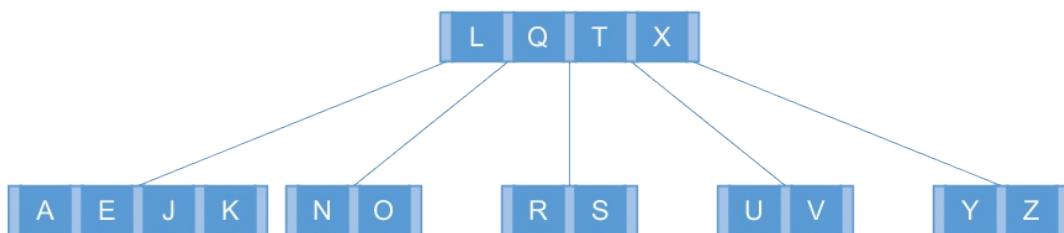
18.3 Deleting a key from a B-tree

18.3-1

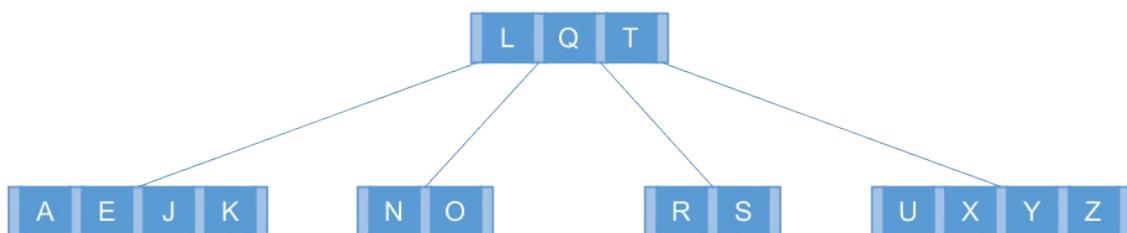
Show the results of deleting C , P , and V , in order, from the tree of Figure 18.8(f).



Initial state



After deleting P



18.3-2

Write pseudocode for B-TREE-DELETE.

...

Problems

18-1 Stacks on secondary storage

Consider implementing a stack in a computer that has a relatively small amount of fast primary memory and a relatively large amount of slower disk storage. The operations PUSH and POP work on single-word values. The stack we wish to support can grow to be much larger than can fit in memory, and thus most of it must be stored on disk.

A simple, but inefficient, stack implementation keeps the entire stack on disk. We maintain in-memory a stack pointer, which is the disk address of the top element on the stack. If the pointer has value p , the top element is the $(p \bmod m)$ th word on page $\lfloor p/m \rfloor$ of the disk, where m is the number of words per page.

To implement the PUSH operation, we increment the stack pointer, read the appropriate page into memory from disk, copy the element to be pushed to the appropriate word on the page, and write the page back to disk. A POP operation is similar. We decrement the stack pointer, read in the appropriate page from disk, and return the top of the stack. We need not write back the page, since it was not modified.

Because disk operations are relatively expensive, we count two costs for any implementation: the total number of disk accesses and the total CPU time. Any disk access to a page of m words incurs charges of one disk access and $\Theta(m)$ CPU time.

a. Asymptotically, what is the worst-case number of disk accesses for n stack operations using this simple implementation? What is the CPU time for n stack operations? (Express your answer in terms of m and n for this and subsequent parts.)

Worst-case number of disk accesses: n read + n write.

CPU time: $\Theta(nm)$.

Now consider a stack implementation in which we keep one page of the stack in memory. (We also maintain a small amount of memory to keep track of which page is currently in memory.) We can perform a stack operation only if the relevant disk page resides in memory. If necessary, we can write the page currently in memory to the disk and read in the new page from the disk to memory. If the relevant disk page is already in memory, then no disk accesses are required.

b. What is the worst-case number of disk accesses required for n PUSH operations? What is the CPU time?

Worst-case number of disk accesses: $\lceil n/m \rceil + 1$ write.

CPU time: $(\lceil n/m \rceil + 1) * \Theta(m) + n = \Theta(n)$.

c. What is the worst-case number of disk accesses required for n stack operations?

What is the CPU time?

Worst-case number of disk accesses: $\lceil n/2 \rceil$ read + $\lceil n/2 \rceil$ write.

CPU time: $\Theta(nm)$.

Suppose that we now implement the stack by keeping two pages in memory (in addition to a small number of words for bookkeeping).

d. Describe how to manage the stack pages so that the amortized number of disk accesses for any stack operation is $O(1/m)$ and the amortized CPU time for any stack operation is $O(1)$.

Less than $m/3$: load prev page.

Larger than $2m/3$: load next page.

18-2 Joining and splitting 2-3-4 trees

The **join** operation takes two dynamic sets S' and S'' and an element x such that for any $x' \in S'$ and $x'' \in S''$, we have $x'.key < x.key < x''.key$. It returns a set $S = S' \cup \{x\} \cup S''$. The **split** operation is like an "inverse" join: given a dynamic set S and an element $x \in S$, it creates a set S' that consists of all elements in set S and an element $x \in S$, it creates a set S' that consists of all elements in $S - \{x\}$ whose keys are less than $x.key$ and a set S'' that consists of all elements in $S - \{x\}$ whose keys are greater than $x.key$. In this problem, we investigate how to implement these operations on 2-3-4 trees. We assume for convenience that elements consist only of keys and that all key values are distinct.

- a.** Show how to maintain, for every node x of a 2-3-4 tree, the height of the subtree rooted at x as an attribute $x.height$. Make sure that your implementation does not affect the asymptotic running times of searching, insertion, and deletion.
- b.** Show how to implement the join operation. Given two 2-3-4 trees T' and T'' and a key k , the join operation should run in $O(1 + |h' - h''|)$ time, where h' and h'' are the heights of T' and T'' , respectively.
- c.** Consider the simple path p from the root of a 2-3-4 tree T to a given key k , the set S' of keys in T that are less than k , and the set S'' of keys in T that are greater than k . Show that p breaks S' into a set of trees $\{T'_0, T'_1, \dots, T'_m\}$ and a set of keys $\{k'_1, k'_2, \dots, k'_m\}$, where, for $i = 1, 2, \dots, m$, we have $y < k'_i < z$ for any keys $y \in T'_{i-1}$ and $z \in T'_i$. What is the relationship between the heights of T'_{i-1} and T'_i ? Describe how p breaks S'' into sets of trees and keys.
- d.** Show how to implement the split operation on T . Use the join operation to assemble the keys in S' into a single 2-3-4 tree T' and the keys in S'' into a single 2-3-4 tree T'' . The running time of the split operation should be $O(\lg n)$, where n is the number of keys in T . (Hint: The costs for joining should telescope.)

19 Fibonacci Heaps

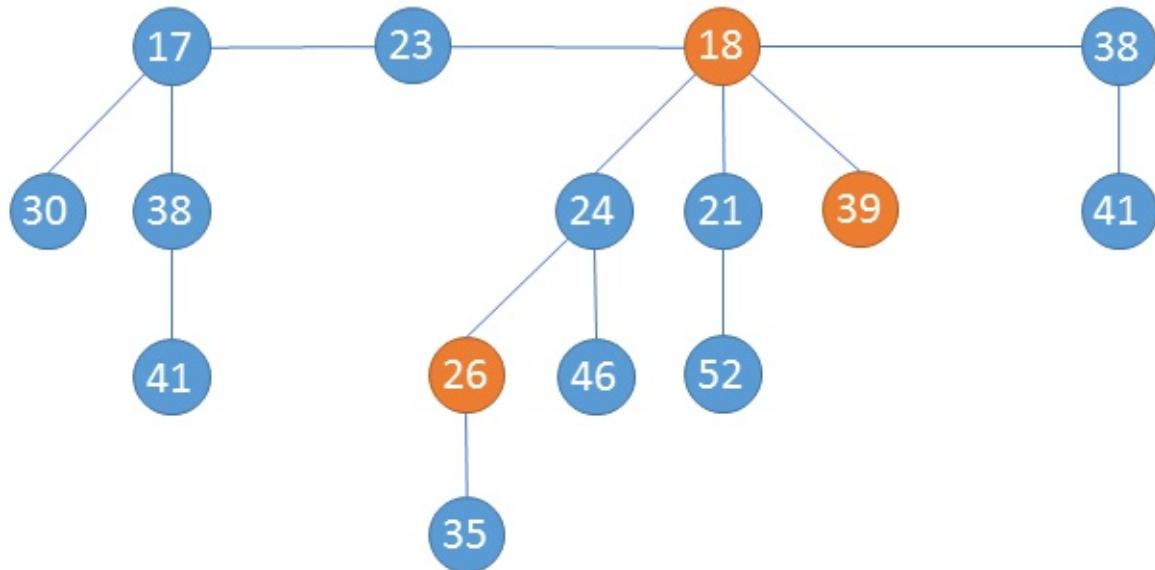
- 19.1 Structure of Fibonacci heaps
- 19.2 Mergeable-heap operations
- 19.3 Decreasing a key and deleting a node
- 19.4 Bounding the maximum degree
- Problems

19.1 Structure of Fibonacci heaps

19.2 Mergeable-heap operations

19.2-1

Show the Fibonacci heap that results from calling FIB-HEAP-EXTRACT-MIN on the Fibonacci heap shown in Figure 19.4(m).



19.3 Decreasing a key and deleting a node

19.3-1

Suppose that a root x in a Fibonacci heap is marked. Explain how x came to be a marked root. Argue that it doesn't matter to the analysis that x is marked, even though it is not a root that was first linked to another node and then lost one child.

19.3-2

Justify the $O(1)$ amortized time of FIB-HEAP-DECREASE-KEY as an average cost per operation by using aggregate analysis.

19.4 Bounding the maximum degree

19.4-1

Professor Pinocchio claims that the height of an n -node Fibonacci heap is $O(\lg n)$. Show that the professor is mistaken by exhibiting, for any positive integer n , a sequence of Fibonacci-heap operations that creates a Fibonacci heap consisting of just one tree that is a linear chain of n nodes.

Initialize: insert 3 numbers then extract-min.

Iteration: insert 3 numbers, in which at least two numbers are less than the root of chain, then extract-min. The smallest newly inserted number will be extracted and the remaining two numbers will form a heap whose degree of root is 1, and since the root of the heap is less than the old chain, the chain will be merged into the newly created heap. Finally we should delete the node which contains the largest number of the 3 inserted numbers.

19.4-2

Suppose we generalize the cascading-cut rule to cut a node x from its parent as soon as it loses its k th child, for some integer constant k . (The rule in Section 19.3 uses $k = 2$.) For what values of k is $D(n) = O(\lg n)$?

Problems

19-1 Alternative implementation of deletion

Professor Pisano has proposed the following variant of the FIB-HEAP-DELETE procedure, claiming that it runs faster when the node being deleted is not the node pointed to by $H.\min$.

```
PISANO-DELETE(H, x)
1 if x == H.min
2     FIB-HEAP-EXTRACT-MIN(H)
3 else y = x.p
4     if y != NIL
5         CUT(H, x, y)
6         CASCADING-CUT(H, y)
7     add x's child list to the root list of H
8     remove x from the root list of H
```

- a.** The professor's claim that this procedure runs faster is based partly on the assumption that line 7 can be performed in $O(1)$ actual time. What is wrong with this assumption?

The largest degree is $D(n) = O(\lg n)$

- b.** Give a good upper bound on the actual time of PISANO-DELETE when x is not $H.\min$. Your bound should be in terms of $x.degree$ and the number c of calls to the CASCADING-CUT procedure.

$O(x.degree + c)$

- c.** Suppose that we call PISANO-DELETE (H, x) , and let H' be the Fibonacci heap that results. Assuming that node x is not a root, bound the potential of H' in terms of $x.degree$, c , $t(H)$, and $m(H)$.

$$\Phi(H') = [t(H) + x.degree + c] + 2[m(H) - c + 2]$$

- d.** Conclude that the amortized time for PISANO-DELETE is asymptotically no better than for FIB-HEAP-DELETE, even when $x \neq H.\min$.

$O(x.\text{degree} + c) + x.\text{degree} + 4 - c = O(x.\text{degree} + c) = O(\lg n)$ is worse than $O(1)$.

19-2 Binomial trees and binomial heaps

The **binomial tree** B_k is an ordered tree (see Section B.5.2) defined recursively. As shown in Figure 19.6(a), the binomial tree B_0 consists of a single node. The binomial tree B_k consists of two binomial trees B_{k-1} that are linked together so that the root of one is the leftmost child of the root of the other. Figure 19.6(b) shows the binomial trees B_0 through B_4 .

a. Show that for the binomial tree B_k ,

1. there are 2^k nodes,
 2. the height of the tree is k ,
 3. there are exactly $\binom{k}{i}$ nodes at depth i for $i = 0, 1, \dots, k$, and
 4. the root has degree k , which is greater than that of any other node; moreover, as Figure 19.6(c) shows, if we number the children of the root from left to right by $k-1, k-2, \dots, 0$, then child i is the root of a subtree B_i .
1. B_k consists of two binomial trees B_{k-1} .
 2. The height of one B_{k-1} is increased by 1.
 3. For $i = 0$, $\binom{k}{0} = 1$ and only root is at depth 0. Suppose in B_{k-1} , the number of nodes at depth i is $\binom{k-1}{i}$, in B_k , the number of nodes at depth i is $\binom{k-1}{i} + \binom{k-1}{i-1} = \binom{k}{i}$.
 4. The degree of the root increase by 1.

A **binomial heap** H is a set of binomial trees that satisfies the following properties:

1. Each node has a key (like a Fibonacci heap).
 2. Each binomial tree in H obeys the min-heap property.
 3. For any nonnegative integer k , there is at most one binomial tree in H whose root has degree k .
- b.** Suppose that a binomial heap H has a total of n nodes. Discuss the relationship between the binomial trees that H contains and the binary representation of n . Conclude that H consists of at most $\lfloor \lg n \rfloor + 1$ binomial trees.

The same as the binary representation of n .

Suppose that we represent a binomial heap as follows. The left-child, right-sibling scheme of Section 10.4 represents each binomial tree within a binomial heap. Each node contains its key; pointers to its parent, to its leftmost child, and to the sibling immediately to its right (these pointers are NIL when appropriate); and its degree (as in Fibonacci heaps, how many children it has). The roots form a singly linked root list, ordered by the degrees of the roots (from low to high), and we access the binomial heap by a pointer to the first node on the root list.

- c. Complete the description of how to represent a binomial heap (i.e., name the attributes, describe when attributes have the value NIL, and define how the root list is organized), and show how to implement the same seven operations on binomial heaps as this chapter implemented on Fibonacci heaps. Each operation should run in $O(\lg n)$ worst-case time, where n is the number of nodes in the binomial heap (or in the case of the UNION operation, in the two binomial heaps that are being united). The MAKE-HEAP operation should take constant time.
- d. Suppose that we were to implement only the mergeable-heap operations on a Fibonacci heap (i.e., we do not implement the DECREASE-KEY or DELETE operations). How would the trees in a Fibonacci heap resemble those in a binomial heap? How would they differ? Show that the maximum degree in an n -node Fibonacci heap would be at most $\lfloor \lg n \rfloor$.
- e. Professor McGee has devised a new data structure based on Fibonacci heaps. A McGee heap has the same structure as a Fibonacci heap and supports just the mergeable-heap operations. The implementations of the operations are the same as for Fibonacci heaps, except that insertion and union consolidate the root list as their last step. What are the worst-case running times of operations on McGee heaps?

19-3 More Fibonacci-heap operations

We wish to augment a Fibonacci heap H to support two new operations without changing the amortized running time of any other Fibonacci-heap operations.

- a. The operation FIB-HEAP-CHANGE-KEY (H, x, k) changes the key of node x to the value k . Give an efficient implementation of FIB-HEAP-CHANGE-KEY, and analyze the amortized running time of your implementation for the cases in which k is greater than, less than, or equal to $x.key$.
- b. Give an efficient implementation of FIB-HEAP-PRUNE (H, r) , which deletes $q = \min(r, H.n)$ nodes from H . You may choose any q nodes to delete. Analyze the amortized running time of your implementation. (Hint: You may need to modify the data structure and potential function.)

19-4 2-3-4 heaps

Chapter 18 introduced the 2-3-4 tree, in which every internal node (other than possibly the root) has two, three, or four children and all leaves have the same depth. In this problem, we shall implement **2-3-4 heaps**, which support the mergeable-heap operations.

The 2-3-4 heaps differ from 2-3-4 trees in the following ways. In 2-3-4 heaps, only leaves store keys, and each leaf x stores exactly one key in the attribute $x.key$. The keys in the leaves may appear in any order. Each internal node x contains a value $x.small$ that is equal to the smallest key stored in any leaf in the subtree rooted at x . The root r contains an attribute $r.height$ that gives the height of the tree. Finally, 2-3-4 heaps are designed to be kept in main memory, so that disk reads and writes are not needed.

Implement the following 2-3-4 heap operations. In parts (a)–(e), each operation should run in $O(\lg n)$ time on a 2-3-4 heap with n elements. The UNION operation in part (f) should run in $O(\lg n)$ time, where n is the number of elements in the two input heaps.

- a. MINIMUM, which returns a pointer to the leaf with the smallest key.

Choose the smallest child in each layer.

- b. DECREASE-KEY, which decreases the key of a given leaf x to a given value $k \leq x.key$.

Decrease the key and update $x.small$ upwards to the root.

- c. INSERT, which inserts leaf x with key k .

Insert just like the insertion in B-trees and update $x.\text{small}$ is the inserted key is less than $x.\text{small}$.

| **d.** DELETE, which deletes a given leaf x .

Delete the key and recalculate $x.\text{small}$ upwards. Since an internal node has at most 4 children, which is a constant, it still runs in $O(\lg n)$.

| **e.** EXTRACT-MIN, which extracts the leaf with the smallest key.

MINIMUM and DELETE.

| **f.** UNION, which unites two 2-3-4 heaps, returning a single 2-3-4 heap and destroying the input heaps.

Based on problem 18-2.

20 van Emde Boas Trees

- 20.1 Preliminary approaches
- 20.2 A recursive structure
- 20.3 The van Emde Boas tree
- Problems

20.1 Preliminary approaches

20.1-1

Modify the data structures in this section to support duplicate keys.

Bit vector => integer vector.

20.1-2

Modify the data structures in this section to support keys that have associated satellite data.

Satellite area.

20.1-3

Observe that, using the structures in this section, the way we find the successor and predecessor of a value x does not depend on whether x is in the set at the time. Show how to find the successor of x in a binary search tree when x is not stored in the tree.

For each node, if $x \leq \text{node.key}$, then node.key is a candidate and descend to the node's left child; otherwise do nothing and descend to the node's right child. There are at most $\lg n$ candidates, the successor is the minimal candidate, and the method runs in $\Theta(\lg n)$.

20.1-4

Suppose that instead of superimposing a tree of degree \sqrt{u} , we were to superimpose a tree of degree $u^{1/k}$, where $k > 1$ is a constant. What would be the height of such a tree, and how long would each of the operations take?

Height: k

How long: $O(k\sqrt{u})$

20.2 A recursive structure

20.2-1

Write pseudocode for the procedures PROTO-VEB-MAXIMUM and PROTO-VEB-PREDECESSOR.

```

import math

class ProtoVEB:
    def __init__(self, u):
        self.u = u
        self.sqrt = int(math.sqrt(u))
        if self.is_leaf():
            self.a = [0, 0]
        else:
            self.summary = ProtoVEB(self.sqrt)
            self.cluster = []
            for _ in xrange(self.sqrt):
                self.cluster.append(ProtoVEB(self.sqrt))

    def is_leaf(self):
        return self.u == 2

    def high(self, x):
        return x / self.sqrt

    def low(self, x):
        return x % self.sqrt

    def index(self, x, y):
        return x * self.sqrt + y

    def member(self, x):
        if self.is_leaf():
            return self.a[x]
        return self.cluster[self.high(x)].member(self.low(x))

    def minimum(self):
        if self.is_leaf():
            if self.a[0] > 0:
                return 0
            if self.a[1] > 0:
                return 1
            return None
        min_idx = self.summary.minimum()
        if min_idx is None:

```

```

        return None
    offset = self.cluster[min_idx].minimum()
    return self.index(min_idx, offset)

def maximum(self):
    if self.is_leaf():
        if self.a[1] > 0:
            return 1
        if self.a[0] > 0:
            return 0
        return None
    max_idx = self.summary.maximum()
    if max_idx is None:
        return None
    offset = self.cluster[max_idx].maximum()
    return self.index(max_idx, offset)

def predecessor(self, x):
    if self.is_leaf():
        if self.a[0] == 1 and x == 1:
            return 0
        return None
    offset = self.cluster[self.high(x)].predecessor(self.low(x))
    if offset is not None:
        return self.index(self.high(x), offset)
    pred_idx = self.summary.predecessor(self.high(x))
    if pred_idx is None:
        return None
    offset = self.cluster[pred_idx].maximum()
    return self.index(pred_idx, offset)

def successor(self, x):
    if self.is_leaf():
        if x == 0 and self.a[1] == 1:
            return 1
        return None
    offset = self.cluster[self.high(x)].successor(self.low(x))
    if offset is not None:
        return self.index(self.high(x), offset)
    succ_idx = self.summary.successor(self.high(x))
    if succ_idx is None:
        return None
    offset = self.cluster[succ_idx].minimum()
    return self.index(succ_idx, offset)

def insert(self, x):
    if self.is_leaf():
        self.a[x] = 1
    else:
        self.summary.insert(self.high(x))
        self.cluster[self.high(x)].insert(self.low(x))

def display(self, space=0, summary=False):

```

```

if self.is_leaf():
    if summary:
        print(' ' * space + 'S ' + str(self.u) + ' ' + str(self.a))
    else:
        print(' ' * space + 'C ' + str(self.u) + ' ' + str(self.a))
else:
    if summary:
        print(' ' * space + 'S ' + str(self.u))
    else:
        print(' ' * space + 'C ' + str(self.u))
    self.summary.display(space + 2, True)
    for c in self.cluster:
        c.display(space + 2)

```

20.2-2

Write pseudocode for PROTO-VEB-DELETE. It should update the appropriate summary bit by scanning the related bits within the cluster. What is the worst-case running time of your procedure?

```

def delete(self, x):
    if self.is_leaf():
        self.a[x] = 0
    else:
        self.cluster[self.high(x)].delete(self.low(x))
        if self.cluster[self.high(x)].minimum() is None:
            self.summary.delete(self.high(x))

```

$$T(u) = 2T(\sqrt{u}) + \Theta(\lg \sqrt{u}) = \Theta(\lg u \lg \lg u)$$

20.2-3

Add the attribute n to each proto-vEB structure, giving the number of elements currently in the set it represents, and write pseudocode for PROTO-VEB-DELETE that uses the attribute n to decide when to reset summary bits to 0. What is the worst-case running time of your procedure? What other procedures need to change because of the new attribute? Do these changes affect their running times?

```

def insert(self, x):
    if self.is_leaf():
        if self.a[x] == 0:
            self.a[x] = 1
            self.n += 1
            return True
        return False
    new_elem = self.cluster[self.high(x)].insert(self.low(x))
    if new_elem:
        self.n += 1
    self.summary.insert(self.high(x))
    return new_elem

def delete(self, x):
    if self.is_leaf():
        if self.a[x] == 1:
            self.a[x] = 0
            self.n -= 1
            return True
        return False
    del_elem = self.cluster[self.high(x)].delete(self.low(x))
    if del_elem:
        self.n -= 1
    if self.cluster[self.high(x)].n == 0:
        self.summary.delete(self.high(x))
    return del_elem

```

Worst-case: $T(u) = 2T(\sqrt{u}) + O(1) = \Theta(\lg n)$

20.2-4

Modify the proto-vEB structure to support duplicate keys.

```

class ProtoVEB:
    def __init__(self, u):
        self.u = u
        self.n = 0
        self.sqrt = int(math.sqrt(u))
        if self.is_leaf():
            self.a = [0, 0]
        else:
            self.summary = ProtoVEB(self.sqrt)
            self.cluster = []
            for _ in xrange(self.sqrt):
                self.cluster.append(ProtoVEB(self.sqrt))

    def is_leaf(self):
        return self.u == 2

```

```

def high(self, x):
    return x / self.sqrt

def low(self, x):
    return x % self.sqrt

def index(self, x, y):
    return x * self.sqrt + y

def member(self, x):
    if self.is_leaf():
        return self.a[x]
    return self.cluster[self.high(x)].member(self.low(x))

def minimum(self):
    if self.is_leaf():
        if self.a[0] > 0:
            return 0
        if self.a[1] > 0:
            return 1
        return None
    min_idx = self.summary.minimum()
    if min_idx is None:
        return None
    offset = self.cluster[min_idx].minimum()
    return self.index(min_idx, offset)

def maximum(self):
    if self.is_leaf():
        if self.a[1] > 0:
            return 1
        if self.a[0] > 0:
            return 0
        return None
    max_idx = self.summary.maximum()
    if max_idx is None:
        return None
    offset = self.cluster[max_idx].maximum()
    return self.index(max_idx, offset)

def predecessor(self, x):
    if self.is_leaf():
        if self.a[0] > 0 and x == 1:
            return 0
        return None
    offset = self.cluster[self.high(x)].predecessor(self.low(x))
    if offset is not None:
        return self.index(self.high(x), offset)
    pred_idx = self.summary.predecessor(self.high(x))
    if pred_idx is None:
        return None
    offset = self.cluster[pred_idx].maximum()
    return self.index(pred_idx, offset)

```

```

def successor(self, x):
    if self.is_leaf():
        if x == 0 and self.a[1] > 0:
            return 1
        return None
    offset = self.cluster[self.high(x)].successor(self.low(x))
    if offset is not None:
        return self.index(self.high(x), offset)
    succ_idx = self.summary.successor(self.high(x))
    if succ_idx is None:
        return None
    offset = self.cluster[succ_idx].minimum()
    return self.index(succ_idx, offset)

def insert(self, x):
    self.n += 1
    if self.is_leaf():
        self.a[x] += 1
    else:
        self.cluster[self.high(x)].insert(self.low(x))
        self.summary.insert(self.high(x))

def delete(self, x):
    if self.is_leaf():
        if self.a[x] > 0:
            self.a[x] -= 1
            self.n -= 1
            return True
        return False
    del_elem = self.cluster[self.high(x)].delete(self.low(x))
    if del_elem:
        self.n -= 1
        self.summary.delete(self.high(x))
    return del_elem

```

20.2-5

Modify the proto-vEB structure to support keys that have associated satellite data.

```

class ProtoVEB:
    def __init__(self, u):
        self.u = u
        self.n = 0
        self.sqrt = int(math.sqrt(u))
        if self.is_leaf():
            self.a = [0, 0]
            self.data = [None, None]
        else:
            self.summary = ProtoVEB(self.sqrt)
            self.cluster = []

```

```

        for _ in xrange(self.sqrt):
            self.cluster.append(ProtoVEB(self.sqrt))

    def is_leaf(self):
        return self.u == 2

    def high(self, x):
        return x / self.sqrt

    def low(self, x):
        return x % self.sqrt

    def index(self, x, y):
        return x * self.sqrt + y

    def member(self, x):
        if self.is_leaf():
            return self.a[x]
        return self.cluster[self.high(x)].member(self.low(x))

    def get_data(self, x):
        if self.is_leaf():
            return self.data[x]
        return self.cluster[self.high(x)].get_data(self.low(x))

    def minimum(self):
        if self.is_leaf():
            if self.a[0] == 1:
                return 0
            if self.a[1] == 1:
                return 1
            return None
        min_idx = self.summary.minimum()
        if min_idx is None:
            return None
        offset = self.cluster[min_idx].minimum()
        return self.index(min_idx, offset)

    def maximum(self):
        if self.is_leaf():
            if self.a[1] == 1:
                return 1
            if self.a[0] == 1:
                return 0
            return None
        max_idx = self.summary.maximum()
        if max_idx is None:
            return None
        offset = self.cluster[max_idx].maximum()
        return self.index(max_idx, offset)

    def predecessor(self, x):
        if self.is_leaf():

```

```

        if self.a[0] == 1 and x == 1:
            return 0
        return None
    offset = self.cluster[self.high(x)].predecessor(self.low(x))
    if offset is not None:
        return self.index(self.high(x), offset)
    pred_idx = self.summary.predecessor(self.high(x))
    if pred_idx is None:
        return None
    offset = self.cluster[pred_idx].maximum()
    return self.index(pred_idx, offset)

def successor(self, x):
    if self.is_leaf():
        if x == 0 and self.a[1] == 1:
            return 1
        return None
    offset = self.cluster[self.high(x)].successor(self.low(x))
    if offset is not None:
        return self.index(self.high(x), offset)
    succ_idx = self.summary.successor(self.high(x))
    if succ_idx is None:
        return None
    offset = self.cluster[succ_idx].minimum()
    return self.index(succ_idx, offset)

def insert(self, x, data):
    if self.is_leaf():
        if self.a[x] == 0:
            self.a[x] = 1
            self.data[x] = data
            self.n += 1
            return True
        return False
    new_elem = self.cluster[self.high(x)].insert(self.low(x), data)
    if new_elem:
        self.n += 1
    self.summary.insert(self.high(x), None)
    return new_elem

def delete(self, x):
    if self.is_leaf():
        if self.a[x] == 1:
            self.a[x] = 0
            self.data[x] = None
            self.n -= 1
            return True
        return False
    del_elem = self.cluster[self.high(x)].delete(self.low(x))
    if del_elem:
        self.n -= 1
    if self.cluster[self.high(x)].n == 0:
        self.summary.delete(self.high(x))

```

```
return del_elem
```

20.2-6

Write pseudocode for a procedure that creates a proto-vEB (u) structure.

See exercise 20.2-1.

20.2-7

Argue that if line 9 of PROTO-VEB-MINIMUM is executed, then the proto-vEB structure is empty.

Obviously.

20.2-8

Suppose that we designed a proto-vEB structure in which each *cluster* array had only $u^{1/4}$ elements. What would the running times of each operation be?

There are $u^{3/4}$ clusters in each proto-vEB.

MEMBER/INSRT: $T(u) = T(u^{1/4}) + O(1) = \Theta(\lg \log_4 u) = \Theta(\lg \lg u)$

MINIMUM/MAXIMUM: $T(u) = T(u^{1/4}) + T(u^{3/4}) + O(1) = \Theta(\lg u)$

SUCCESSOR/PREDECESSOR/DELETE:

$T(u) = T(u^{1/4}) + T(u^{3/4}) + \Theta(\lg u^{1/4}) = \Theta(\lg u \lg \lg u)$

20.3 The van Emde Boas tree

20.3-1

Modify vEB trees to support duplicate keys.

```

class VanEmdeBoasTree:
    def __init__(self, u):
        self.u = u
        temp = u
        bit_num = -1
        while temp > 0:
            temp >>= 1
            bit_num += 1
        self.sqrt_h = 1 << ((bit_num + 1) // 2)
        self.sqrt_l = 1 << (bit_num // 2)
        self.min = None
        self.max = None
        self.min_cnt = 0
        self.max_cnt = 0
        if not self.is_leaf():
            self.summary = VanEmdeBoasTree(self.sqrt_h)
            self.cluster = []
            for _ in xrange(self.sqrt_h):
                self.cluster.append(VanEmdeBoasTree(self.sqrt_l))

    def is_leaf(self):
        return self.u == 2

    def high(self, x):
        return x / self.sqrt_l

    def low(self, x):
        return x % self.sqrt_l

    def index(self, x, y):
        return x * self.sqrt_l + y

    def minimum(self):
        return self.min

    def maximum(self):
        return self.max

    def member(self, x):
        if x == self.min or x == self.max:
            return True
        if self.is_leaf():
            return False

```

```

        return self.member(self.cluster[self.high(x)], self.low(x))

def successor(self, x):
    if self.is_leaf():
        if x == 0 and self.max == 1:
            return 1
        return None
    if self.min is not None and x < self.min:
        return self.min
    max_low = self.cluster[self.high(x)].maximum()
    if max_low is not None and self.low(x) < max_low:
        offset = self.cluster[self.high(x)].successor(self.low(x))
        return self.index(self.high(x), offset)
    succ_cluster = self.summary.successor(self.high(x))
    if succ_cluster is None:
        return None
    offset = self.cluster[succ_cluster].minimum()
    return self.index(succ_cluster, offset)

def predecessor(self, x):
    if self.is_leaf():
        if x == 1 and self.min == 0:
            return 0
        return None
    if self.max is not None and x > self.max:
        return self.max
    min_low = self.cluster[self.high(x)].minimum()
    if min_low is not None and self.low(x) > min_low:
        offset = self.cluster[self.high(x)].predecessor(self.low(x))
        return self.index(self.high(x), offset)
    pred_cluster = self.summary.predecessor(self.high(x))
    if pred_cluster is None:
        if self.min is not None and x > self.min:
            return self.min
        return None
    offset = self.cluster[pred_cluster].maximum()
    return self.index(pred_cluster, offset)

def insert_empty(self, x, n):
    self.min = x
    self.max = x
    self.min_cnt = self.max_cnt = n

def insert(self, x, n=1):
    if self.min is None:
        self.insert_empty(x, n)
        return
    if x == self.max:
        self.max_cnt += n
    if x == self.min:
        self.min_cnt += n
        return
    if x < self.min:

```

```

        x, self.min = self.min, x
        n, self.min_cnt = self.min_cnt, n
    if not self.is_leaf():
        if self.cluster[self.high(x)].minimum() is None:
            self.summary.insert(self.high(x))
            self.cluster[self.high(x)].insert_empty(self.low(x), n)
        else:
            self.cluster[self.high(x)].insert(self.low(x), n)
    if x > self.max:
        self.max = x
        self.max_cnt = n

def delete(self, x, n=1):
    if self.min == self.max:
        if self.min is None or self.min_cnt == n:
            self.min = self.max = None
            self.min_cnt = 0
        else:
            self.min_cnt -= n
            self.max_cnt = self.min_cnt
        return
    if self.is_leaf():
        if x == 0:
            self.min_cnt -= n
            if self.min_cnt == 0:
                self.min = 1
                self.min_cnt = self.max_cnt
            else:
                self.max_cnt -= n
                if self.max_cnt == 0:
                    self.max = 0
                    self.max_cnt = self.min_cnt
        return
    next_n = n
    if x == self.min:
        if self.min_cnt > n:
            self.min_cnt -= n
            return
        first_cluster = self.summary.minimum()
        x = self.index(first_cluster,
                       self.cluster[first_cluster].minimum())
        self.min = x
        self.min_cnt = self.cluster[first_cluster].min_cnt
        next_n = self.cluster[first_cluster].min_cnt
    self.cluster[self.high(x)].delete(self.low(x), next_n)
    if self.cluster[self.high(x)].minimum() is None:
        self.summary.delete(self.high(x))
    if x == self.max:
        if self.max == self.min:
            self.max_cnt = self.min_cnt
        self.max_cnt -= n
        if self.max_cnt == 0:

```

```

        sum_max = self.summary.maximum()
        if sum_max is None:
            self.max = self.min
            self.max_cnt = self.min_cnt
        else:
            self.max = self.index(sum_max,
                                  self.cluster[sum_max].maximum())
            self.max_cnt = self.cluster[sum_max].max_cnt
    elif x == self.max:
        if self.max == self.min:
            self.max_cnt = self.min_cnt
        return
    self.max_cnt -= 1
    if self.max_cnt == 0:
        self.max = self.index(self.high(x),
                              self.cluster[self.high(x)].maximum())
        self.max_cnt = self.cluster[self.high(x)].max_cnt

def display(self, space=0, summary=False):
    disp = ' ' * space
    if summary:
        disp += 'S '
    else:
        disp += 'C '
    disp += str(self.u) + ' ' + str(self.min) + ' ' + str(self.max) + ' | '
    disp += str(self.min_cnt) + ' ' + str(self.max_cnt)
    print(disp)
    if not self.is_leaf():
        self.summary.display(space + 2, True)
        for c in self.cluster:
            c.display(space + 2)

```

20.3-2

Modify vEB trees to support keys that have associated satellite data.

```

class VanEmdeBoasTree:
    def __init__(self, u):
        self.u = u
        temp = u
        bit_num = -1
        while temp > 0:
            temp >>= 1
            bit_num += 1
        self.sqrt_h = 1 << ((bit_num + 1) // 2)
        self.sqrt_l = 1 << (bit_num // 2)
        self.min = None
        self.max = None
        self.min_data = None
        self.max_data = None
        if not self.is_leaf():

```

```

        self.summary = VanEmdeBoasTree(self.sqrt_h)
        self.cluster = []
        for _ in xrange(self.sqrt_h):
            self.cluster.append(VanEmdeBoasTree(self.sqrt_l))

    def is_leaf(self):
        return self.u == 2

    def high(self, x):
        return x / self.sqrt_l

    def low(self, x):
        return x % self.sqrt_l

    def index(self, x, y):
        return x * self.sqrt_l + y

    def minimum(self):
        return self.min

    def maximum(self):
        return self.max

    def member(self, x):
        if x == self.min or x == self.max:
            return True
        if self.is_leaf():
            return False
        return self.member(self.cluster[self.high(x)], self.low(x))

    def get_data(self, x):
        if x == self.min:
            return self.min_data
        if x == self.max:
            return self.max_data
        if self.is_leaf():
            return None
        return self.cluster[self.high(x)].get_data(self.low(x))

    def successor(self, x):
        if self.is_leaf():
            if x == 0 and self.max == 1:
                return 1
            return None
        if self.min is not None and x < self.min:
            return self.min
        max_low = self.cluster[self.high(x)].maximum()
        if max_low is not None and self.low(x) < max_low:
            offset = self.cluster[self.high(x)].successor(self.low(x))
            return self.index(self.high(x), offset)
        succ_cluster = self.summary.successor(self.high(x))
        if succ_cluster is None:
            return None

```

```

        offset = self.cluster[succ_cluster].minimum()
        return self.index(succ_cluster, offset)

def predecessor(self, x):
    if self.is_leaf():
        if x == 1 and self.min == 0:
            return 0
        return None
    if self.max is not None and x > self.max:
        return self.max
    min_low = self.cluster[self.high(x)].minimum()
    if min_low is not None and self.low(x) > min_low:
        offset = self.cluster[self.high(x)].predecessor(self.low(x))
        return self.index(self.high(x), offset)
    pred_cluster = self.summary.predecessor(self.high(x))
    if pred_cluster is None:
        if self.min is not None and x > self.min:
            return self.min
        return None
    offset = self.cluster[pred_cluster].maximum()
    return self.index(pred_cluster, offset)

def insert_empty(self, x, data):
    self.min = x
    self.max = x
    self.min_data = self.max_data = data

def insert(self, x, data):
    if self.min is None:
        self.insert_empty(x, data)
    else:
        if x < self.min:
            x, self.min = self.min, x
            data, self.min_data = self.min_data, data
        if not self.is_leaf():
            if self.cluster[self.high(x)].minimum() is None:
                self.summary.insert(self.high(x), data)
                self.cluster[self.high(x)].insert_empty(self.low(x), data)
            else:
                self.cluster[self.high(x)].insert(self.low(x), data)
        if x > self.max:
            self.max = x
            self.max_data = data

def delete(self, x):
    if self.min == self.max:
        self.min = self.max = None
        self.min_data = self.max_data = None
    elif self.is_leaf():
        if x == 0:
            self.min = 1
            self.min_data = self.max_data
    else:

```

```

        self.min = 0
        self.max = self.min
        self.max_data = self.min_data
    else:
        if x == self.min:
            first_cluster = self.summary.minimum()
            x = self.index(first_cluster,
                           self.cluster[first_cluster].minimum())
            self.min = x
            self.min_data = self.cluster[first_cluster].min_data
            self.cluster[self.high(x)].delete(self.low(x))
            if self.cluster[self.high(x)].minimum() is None:
                self.summary.delete(self.high(x))
            if x == self.max:
                sum_max = self.summary.maximum()
                if sum_max is None:
                    self.max = self.min
                    self.max_data = self.min_data
                else:
                    self.max = self.index(sum_max,
                                          self.cluster[sum_max].maximum())
                    self.max_data = self.cluster[sum_max].max_data
        elif x == self.max:
            self.max = self.index(self.high(x),
                                  self.cluster[self.high(x)].maximum())
            self.max_data = self.cluster[self.high(x)].max_data

    def display(self, space=0, summary=False):
        disp = ' ' * space
        if summary:
            disp += 'S '
        else:
            disp += 'C '
        disp += str(self.u) + ' ' + str(self.min) + ' ' + str(self.max)
        print(disp)
        if not self.is_leaf():
            self.summary.display(space + 2, True)
            for c in self.cluster:
                c.display(space + 2)

```

20.3-3

Write pseudocode for a procedure that creates an empty van Emde Boas tree.

See exercise 20.3-1 and exercise 20.3-2.

20.3-4

What happens if you call VEB-TREE-INSERT with an element that is already in the vEB tree? What happens if you call VEB-TREE-DELETE with an element that is not in the vEB tree? Explain why the procedures exhibit the behavior that they do. Show how to modify vEB trees and their operations so that we can check in constant time whether an element is present.

Already/not: nothing changes.

Constant time: add an auxiliary array of size u .

20.3-5

Suppose that instead of \sqrt{u} clusters, each with universe size \sqrt{u} , we constructed vEB trees to have $u^{1/k}$ clusters, each with universe size $u^{1-1/k}$, where $k > 1$ is a constant. If we were to modify the operations appropriately, what would be their running times? For the purpose of analysis, assume that $u^{1/k}$ and $u^{1-1/k}$ are always integers.

MINIMUM/MAXIMUM: $O(1)$.

SUCCESSOR/PREDECESSOR/INSERT/DELETE worst:

$$T(u) = T(u^{1/k}) + O(1) = O(\lg \log_{1/k} u)$$

20.3-6

Creating a vEB tree with universe size u requires $O(u)$ time. Suppose we wish to explicitly account for that time. What is the smallest number of operations n for which the amortized time of each operation in a vEB tree is $O(\lg \lg u)$?

Since MINIMUM/MAXIMUM is $O(1)$, we need about $\left\lceil \frac{u}{\lg \lg u} \right\rceil$ operations.

Problems

20-1 Space requirements for van Emde Boas trees

This problem explores the space requirements for van Emde Boas trees and suggests a way to modify the data structure to make its space requirement depend on the number n of elements actually stored in the tree, rather than on the universe size u . For simplicity, assume that \sqrt{u} is always an integer.

a. Explain why the following recurrence characterizes the space requirement $P(u)$ of a van Emde Boas tree with universe size u :

$$P(u) = (\sqrt{u} + 1)P(\sqrt{u}) + \Theta(\sqrt{u})$$

\sqrt{u} : number of clusters.

1 : number of summary.

$P(\sqrt{u})$: space of cluster/summary.

$\Theta(\sqrt{u})$: pointers of clusters.

b. Prove that recurrence (20.5) has the solution $P(u) = O(u)$.

Suppose $P(u) \leq cu - d$,

$$\begin{aligned} P(u) &= (\sqrt{u} + 1)P(\sqrt{u}) + \Theta(\sqrt{u}) \\ &\leq (\sqrt{u} + 1) \cdot c \cdot (\sqrt{u} - d) + \Theta(\sqrt{u}) \\ &= cu + c(1 - d)\sqrt{u} - cd + \Theta(\sqrt{u}) \quad (d > 1) \\ &\leq cu \end{aligned}$$

In order to reduce the space requirements, let us define a *reduced-space van Emde Boas tree*, or **RS-vEB tree**, as a **vEB tree** V but with the following changes:

- The attribute $V.\text{cluster}$, rather than being stored as a simple array of pointers to vEB trees with universe size \sqrt{u} , is a hash table (see Chapter 11) stored as a dynamic table (see Section 17.4). Corresponding to the array version of $V.\text{cluster}$, the hash table stores pointers to RS-vEB trees with universe size \sqrt{u} . To find the i th cluster, we look up the key i in the hash table, so that we can find the i th cluster by a single search in the hash table.
- The hash table stores only pointers to nonempty clusters. A search in the hash table for an empty cluster returns NIL, indicating that the cluster is empty.
- The attribute $V.\text{summary}$ is NIL if all clusters are empty. Otherwise, $V.\text{summary}$ points to an RS-vEB tree with universe size \sqrt{u} .

Because the hash table is implemented with a dynamic table, the space it requires is proportional to the number of nonempty clusters.

When we need to insert an element into an empty RS-vEB tree, we create the RS-vEB tree by calling the following procedure, where the parameter u is the universe size of the RS-vEB tree:

```

CREATE-NEW-RS-VEB-TREE( $u$ )
1 allocate a new vEB tree  $V$ 
2  $V.u = u$ 
3  $V.\text{min} = \text{NIL}$ 
4  $V.\text{max} = \text{NIL}$ 
5  $V.\text{summary} = \text{NIL}$ 
6 create  $V.\text{cluster}$  as an empty dynamic hash table
7 return  $V$ 
```

- c. Modify the VEB-TREE-INSERT procedure to produce pseudocode for the procedure $\text{RS-VEB-TREE-INSERT}(V, x)$, which inserts x into the RS-vEB tree V , calling CREATE-NEW-RS-VEB-TREE as appropriate.

```

def get_cluster(self, x):
    if self.cluster[x] is None:
        self.cluster[x] = VanEmdeBoasTree(self.sqrt_l)
    return self.cluster[x]

def get_summary(self):
    if self.summary is None:
        self.summary = VanEmdeBoasTree(self.sqrt_h)
    return self.summary

def insert(self, x):
    if self.min is None:
        self.insert_empty(x)
    else:
        if x < self.min:
            x, self.min = self.min, x
        if not self.is_leaf():
            if self.get_cluster(self.high(x)).minimum() is None:
                self.get_summary().insert(self.high(x))
                self.get_cluster(self.high(x)).insert_empty(self.low(x))
            else:
                self.get_cluster(self.high(x)).insert(self.low(x))
        if x > self.max:
            self.max = x

```

- d. Modify the VEB-TREE-SUCCESSOR procedure to produce pseudocode for the procedure RS-VEB-TREE-SUCCESSOR (V, x) , which returns the successor of x in RS-vEB tree V , or NIL if x has no successor in V .

```

def successor(self, x):
    if self.is_leaf():
        if x == 0 and self.max == 1:
            return 1
        return None
    if self.min is not None and x < self.min:
        return self.min
    max_low = self.get_cluster(self.high(x)).maximum()
    if max_low is not None and self.low(x) < max_low:
        offset = self.get_cluster(self.high(x)).successor(self.low(x))
        return self.index(self.high(x), offset)
    succ_cluster = self.get_summary().successor(self.high(x))
    if succ_cluster is None:
        return None
    offset = self.cluster[succ_cluster].minimum()
    return self.index(succ_cluster, offset)

```

- e. Prove that, under the assumption of simple uniform hashing, your RS-VEBTREE-INSERT and RS-VEB-TREE-SUCCESSOR procedures run in $O(\lg \lg u)$ expected time.

The hashing tasks about $\Theta(1)$ time, thus the procedures run in $O(\lg \lg u)$.

- f. Assuming that elements are never deleted from a vEB tree, prove that the space requirement for the RS-vEB tree structure is $O(n)$, where n is the number of elements actually stored in the RS-vEB tree.

- g. RS-vEB trees have another advantage over vEB trees: they require less time to create. How long does it take to create an empty RS-vEB tree?

$\Theta(\sqrt{u})$ to create the hash table.

20-2 y-fast tries

This problem investigates D. Willard's "y-fast tries" which, like van Emde Boas trees, perform each of the operations MEMBER, MINIMUM, MAXIMUM, PREDECESSOR, and SUCCESSOR on elements drawn from a universe with size u in $O(\lg \lg u)$ worst-case time. The INSERT and DELETE operations take $O(\lg \lg u)$ amortized time. Like reduced-space van Emde Boas trees (see Problem 20-1), yfast tries use only $O(n)$ space to store n elements. The design of y-fast tries relies on perfect hashing (see Section 11.5).

As a preliminary structure, suppose that we create a perfect hash table containing not only every element in the dynamic set, but every prefix of the binary representation of every element in the set. For example, if $u = 16$, so that $\lg u = 4$, and $x = 13$ is in the set, then because the binary representation of 13 is 1101, the perfect hash table would contain the strings 1, 11, 110, and 1101. In addition to the hash table, we create a doubly linked list of the elements currently in the set, in increasing order.

- a. How much space does this structure require?

$O(n \lg u)$

- b.** Show how to perform the MINIMUM and MAXIMUM operations in $O(1)$ time; the MEMBER, PREDECESSOR, and SUCCESSOR operations in $O(\lg \lg u)$ time; and the INSERT and DELETE operations in $O(\lg u)$ time.

To reduce the space requirement to $O(n)$, we make the following changes to the data structure:

- We cluster the n elements into $n/\lg u$ groups of size $\lg u$. (Assume for now that $\lg u$ divides n .) The first group consists of the $\lg u$ smallest elements in the set, the second group consists of the next $\lg u$ smallest elements, and so on.
- We designate a "representative" value for each group. The representative of the i th group is at least as large as the largest element in the i th group, and it is smaller than every element of the $(i + 1)$ st group. (The representative of the last group can be the maximum possible element $u - 1$.) Note that a representative might be a value not currently in the set.
- We store the $\lg u$ elements of each group in a balanced binary search tree, such as a red-black tree. Each representative points to the balanced binary search tree for its group, and each balanced binary search tree points to its group's representative.
- The perfect hash table stores only the representatives, which are also stored in a doubly linked list in increasing order.

We call this structure a **y-fast trie**.

- c.** Show that a y-fast trie requires only $O(n)$ space to store n elements.

The doubly linked list has less than n elements, while the binary search trees contains n nodes, thus a y-fast trie requires $O(n)$ space.

- d.** Show how to perform the MINIMUM and MAXIMUM operations in $O(\lg \lg u)$ time with a y-fast trie.

MINIMUM: Find the minimum representative in the doubly linked list in $\Theta(1)$, then find the minimum element in the binary search tree in $O(\lg \lg u)$.

- e.** Show how to perform the MEMBER operation in $O(\lg \lg u)$ time.

Find the smallest representative greater than k with binary searching in $\Theta(\lg \lg u)$, find the element in the binary search tree in $O(\lg \lg u)$.

f. Show how to perform the PREDECESSOR and SUCCESSOR operations in $O(\lg \lg u)$ time.

SUCCESSOR: Find the smallest representative greater than k with binary searching in $\Theta(\lg \lg u)$, then find whether there is an element in this cluster that is larger than k in $O(\lg \lg u)$. If there is one element greater than k in the representative's cluster, then the successor is in the next representative's cluster, we can locate the next representative with the doubly linked list in $\Theta(1)$.

g. Explain why the INSERT and DELETE operations take $\Omega(\lg \lg u)$ time.

Same as **e**, we need to find the cluster in $\Theta(\lg \lg u)$, then the operations in the binary search tree takes $O(\lg \lg u)$.

h. Show how to relax the requirement that each group in a y-fast trie has exactly $\lg u$ elements to allow INSERT and DELETE to run in $O(\lg \lg u)$ amortized time without affecting the asymptotic running times of the other operations.

Fixed representatives.

21 Data Structures for Disjoint Sets

- 21.1 Disjoint-set operations
- 21.2 Linked-list representation of disjoint sets
- 21.3 Disjoint-set forests
- 21.4 Analysis of union by rank with path compression
- Problems

21.1 Disjoint-set operations

21.1-1

Suppose that CONNECTED-COMPONENTS is run on the undirected graph $G = (V, E)$, where $V = \{a, b, c, d, e, f, g, h, i, j, k\}$ and the edges of E are processed in the order $(d, i), (f, k), (g, i), (b, g), (a, h), (i, j), (d, k), (b, j), (d, f), (g, j), (a, e)$.

List the vertices in each connected component after each iteration of lines 3–5.

Edge processed	Collection of disjoint sets										
initial sets	{a} {b} {c} {d} {e} {f} {g} {h} {i} {j} {k}										
(d, i)	{a} {b} {c} {d, i} {e} {f} {g} {h} {j} {k}										
(f, k)	{a} {b} {c} {d, i} {e} {f, k} {g} {h} {j}										
(g, i)	{a} {b} {c} {d, i} {e} {f, k} {g} {h} {j}										
(b, g)	{a} {b, g} {c} {d, i} {e} {f, k} {h} {j}										
(a, h)	{a, h} {b, g} {c} {d, i} {e} {f, k} {j}										
(i, j)	{a, h} {b, g} {c} {d, i, j} {e} {f, k}										
(d, k)	{a, h} {b, g} {c} {d, f, i, j, k} {e}										
(b, j)	{a, h} {b, d, f, g, i, j, k} {c} {e}										
(d, f)	{a, h} {b, d, f, g, i, j, k} {c} {e}										
(g, j)	{a, h} {b, d, f, g, i, j, k} {c} {e}										
(a, e)	{a, e, h} {b, d, f, g, i, j, k} {c}										

21.1-2

Show that after all edges are processed by CONNECTED-COMPONENTS, two vertices are in the same connected component if and only if they are in the same set.

...

21.1-3

During the execution of CONNECTED-COMPONENTS on an undirected graph $G = (V, E)$ with k connected components, how many times is FIND-SET called? How many times is UNION called? Express your answers in terms of $|V|$, $|E|$, and k .

FIND-SET: $2|E|$.

UNION: Initially, there are $|V|$ components, and each UNION operation decreases the number of connected components by 1, thus UNION is called $|V| - k$ times.

20.2 Linked-list representation of disjoint sets

21.2-1

Write pseudocode for MAKE-SET, FIND-SET, and UNION using the linked-list representation and the weighted-union heuristic. Make sure to specify the attributes that you assume for set objects and list objects.

UNION: if > swap & size +=

21.2-2

Show the data structure that results and the answers returned by the FIND-SET operations in the following program. Use the linked-list representation with the weighted-union heuristic.

```

1  for i = 1 to 16
2      MAKE-SET(x_{i})
3  for i = 1 to 15 by 2
4      UNION(x_{i}, x_{i+1})
5  for i = 1 to 13 by 4
6      UNION(x_{i}, x_{i+2})
7  UNION(x_{1}, x_{5})
8  UNION(x_{11}, x_{13})
9  UNION(x_{1}, x_{10})
10 FIND-SET(x_{2})
11 FIND-SET(x_{9})
```

All same set.

21.2-3

Adapt the aggregate proof of Theorem 21.1 to obtain amortized time bounds of $O(1)$ for MAKE-SET and FIND-SET and $O(\lg n)$ for UNION using the linked-list representation and the weighted-union heuristic.

$$O(n \lg n)/n = O(\lg n)$$

21.2-4

Give a tight asymptotic bound on the running time of the sequence of operations in Figure 21.3 assuming the linked-list representation and the weighted-union heuristic.

$$\Theta(n) + \Theta(n) = \Theta(n)$$

21.2-5

Professor Gompers suspects that it might be possible to keep just one pointer in each set object, rather than two (head and tail), while keeping the number of pointers in each list element at two. Show that the professor's suspicion is well founded by describing how to represent each set by a linked list such that each operation has the same running time as the operations described in this section. Describe also how the operations work. Your scheme should allow for the weighted-union heuristic, with the same effect as described in this section. (Hint: Use the tail of a linked list as its set's representative.)

If each node has a pointer points to the first node and we use the tail as the representative, then we can find the head (first node) with the tail's pointer in $\Theta(1)$.

21.2-6

Suggest a simple change to the UNION procedure for the linked-list representation that removes the need to keep the tail pointer to the last object in each list. Whether or not the weighted-union heuristic is used, your change should not change the asymptotic running time of the UNION procedure. (Hint: Rather than appending one list to another, splice them together.)

We can splice/zip two lists. Suppose one list is $1 \rightarrow 3 \rightarrow 5 \rightarrow \dots \rightarrow 101$, the other is $2 \rightarrow 4 \rightarrow 6$, then the spliced/zipped list is $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 7 \rightarrow \dots \rightarrow 101$. When the shorter one is used up, we can concatenate the remaining part of the longer list directly to the tail of merged list, thus it is identical to the weighted-union heuristic.

21.3 Disjoint-set forests

21.3-1

Redo Exercise 21.2-2 using a disjoint-set forest with union by rank and path compression.

...

21.3-2

Write a nonrecursive version of FIND-SET with path compression.

...

21.3-3

Give a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, that takes $\Omega(m \lg n)$ time when we use union by rank only.

$$\Omega((m - 2n) \lg n) = \Omega(m \lg n)$$

21.3-4

Suppose that we wish to add the operation PRINT-SET (x), which is given a node x and prints all the members of x 's set, in any order. Show how we can add just a single attribute to each node in a disjoint-set forest so that PRINT-SET (x) takes time linear in the number of members of x 's set and the asymptotic running times of the other operations are unchanged. Assume that we can print each member of the set in $O(1)$ time.

Each member has a pointer points to the next element in the set, which forms a circular linked list. When union two sets x and y , swap $x.\text{next}$ and $y.\text{next}$ to merged the two linked lists.

21.3-5 *

Show that any sequence of m MAKE-SET, FIND-SET, and LINK operations, where all the LINK operations appear before any of the FIND-SET operations, takes only $O(m)$ time if we use both path compression and union by rank. What happens in the same situation if we use only the path-compression heuristic?

Suppose that there are n MAKE_SET, then after the LINKs, there are only n elements to compress, thus it takes $O(m)$ time. It doesn't matter whether we use union by rank or not.

21.4 Analysis of union by rank with path compression

21.4-1

| Prove Lemma 21.4.

Obviously.

21.4-2

| Prove that every node has rank at most $\lfloor \lg n \rfloor$.

The rank increases by 1 only when $x.rank = y.rank$, thus each time we need the twice number of elements to increase the rank by 1, therefore the rank is at most $\lfloor \lg n \rfloor$.

21.4-3

| In light of Exercise 21.4-2, how many bits are necessary to store $x.rank$ for each node x ?

$\lceil \lfloor \lg n \rfloor \rceil$

21.4-4

| Using Exercise 21.4-2, give a simple proof that operations on a disjoint-set forest with union by rank but without path compression run in $O(m \lg n)$ time.

$O((m - x) \cdot \lfloor \lg n \rfloor) = O(m \lg n)$

21.4-5

| Professor Dante reasons that because node ranks increase strictly along a simple path to the root, node levels must monotonically increase along the path. In other words, if $x.rank > 0$ and $x.p$ is not a root, then $\text{level}(x) \leq \text{level}(x.p)$. Is the professor correct?

No. Think about an extreme condition $100 \rightarrow 99 \rightarrow 1$, since $\text{level}(99) = 0$, $\text{level}(1) \geq 1$, we have $\text{level}(x) > \text{level}(x.p)$.

21.4-6 *

Consider the function $\alpha'(n) = \min\{k : A_k(1) \geq \lg(n+1)\}$. Show that $\alpha'(n) \leq 3$ for all practical values of n and, using Exercise 21.4-2, show how to modify the potential-function argument to prove that we can perform a sequence of m MAKESET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, on a disjoint-set forest with union by rank and path compression in worst-case time $O(m\alpha'(n))$.

Problems

21-1 Off-line minimum

The **off-line minimum problem** asks us to maintain a dynamic set T of elements from the domain $\{1, 2, \dots, n\}$ under the operations INSERT and EXTRACT-MIN. We are given a sequence S of n INSERT and m EXTRACT-MIN calls, where each key in $\{1, 2, \dots, n\}$ is inserted exactly once. We wish to determine which key is returned by each EXTRACT-MIN call. Specifically, we wish to fill in an array $\text{extracted}[1 \dots m]$, where for $i = 1, 2, \dots, m$, $\text{extracted}[i]$ is the key returned by the i th EXTRACT-MIN call. The problem is "off-line" in the sense that we are allowed to process the entire sequence S before determining any of the returned keys.

- a. In the following instance of the off-line minimum problem, each operation INSERT (i) is represented by the value of i and each EXTRACT-MIN is represented by the letter E:

4, 8, E, 3, E, 9, 2, 6, E, E, 1, 7, E, 5.

Fill in the correct values in the extracted array.

4, 3, 2, 6, 8, 1.

To develop an algorithm for this problem, we break the sequence S into homogeneous subsequences. That is, we represent S by

$I_1, E, I_2, E, I_3, \dots, I_m, E, I_{m+1}$

where each E represents a single EXTRACT-MIN call and each I_j represents a (possibly empty) sequence of INSERT calls. For each subsequence I_j , we initially place the keys inserted by these operations into a set K_j , which is empty if I_j is empty. We then do the following:

```

OFF-LINE-MINIMUM(m, n)
1  for i = 1 to n
2      determine j such that i \in K_j
3      if j \neq m + 1
4          extracted[j] = i
5          let l be the smallest value greater than j
              for which set K_l exists
6          K_l = K_j \cup K_l , destroying K_j
7  return extracted
    
```

b. Argue that the array `extracted` returned by OFF-LINE-MINIMUM is correct.

Greedy.

c. Describe how to implement OFF-LINE-MINIMUM efficiently with a disjoint-set data structure. Give a tight bound on the worst-case running time of your implementation.

```

class DisjointSetForest:
    def __init__(self, n):
        self.p = list(range(n))

    def union(self, x, y):
        self.link(self.find_set(x), self.find_set(y))

    def link(self, x, y):
        self.p[x] = y

    def find_set(self, x):
        if x != self.p[x]:
            self.p[x] = self.find_set(self.p[x])
        return self.p[x]

def off_line_minimum(q, n):
    m = len([0 for v in q if v == 'E'])
    ds = DisjointSetForest(m + 1)
    pos = [-1] * (n + 1)
    i = 0
    for v in q:
        if v == 'E':
            i += 1
        else:
            pos[v] = i
    extracted = [None] * m
    for i in xrange(1, n + 1):
        j = ds.find_set(pos[i])
        if j < m:
            extracted[j] = i
            ds.link(j, j + 1)
    return extracted

```

21-2 Depth determination

In the **depth-determination problem**, we maintain a forest $\mathcal{F} = \{T_i\}$ of rooted trees under three operations:

MAKE-TREE (v) creates a tree whose only node is v .

FIND-DEPTH (v) returns the depth of node v within its tree.

GRAFT (r, v) makes node r , which is assumed to be the root of a tree, become the child of node v , which is assumed to be in a different tree than r but may or may not itself be a root.

a. Suppose that we use a tree representation similar to a disjoint-set forest: $v.p$ is the parent of node v , except that $v.p = v$ if v is a root. Suppose further that we implement **GRAFT** (r, v) by setting $r.p = v$ and **FIND-DEPTH** (v) by following the find path up to the root, returning a count of all nodes other than v encountered. Show that the worst-case running time of a sequence of m **MAKE-TREE**, **FIND-DEPTH**, and **GRAFT** operations is $\Theta(m^2)$.

$m/3$ **MAKE-TREE**, $m/3$ **GRAFT** to make a chain, $m/3$ **FIND-DEPTH**.

By using the union-by-rank and path-compression heuristics, we can reduce the worst-case running time. We use the disjoint-set forest $\mathcal{S} = \{S_i\}$, where each set S_i (which is itself a tree) corresponds to a tree T_i in the forest \mathcal{F} . The tree structure within a set S_i , however, does not necessarily correspond to that of T_i . In fact, the implementation of S_i does not record the exact parent-child relationships but nevertheless allows us to determine any node's depth in T_i .

The key idea is to maintain in each node v a "pseudodistance" $v.d$, which is defined so that the sum of the pseudodistances along the simple path from v to the root of its set S_i equals the depth of v in T_i . That is, if the simple path from v to its root in S_i is v_0, v_1, \dots, v_k , where $v_0 = v$ and v_k is S_i 's root, then the depth of v in T_i is $\sum_{j=0}^k v_j.d$.

b. Give an implementation of **MAKE-TREE**.

```
class TreeNode:
    def __init__(self):
        self.d = 0
        self.p = self
        self.rank = 0
```

- c. Show how to modify FIND-SET to implement FIND-DEPTH. Your implementation should perform path compression, and its running time should be linear in the length of the find path. Make sure that your implementation updates pseudodistances correctly.

```
def find_depth(v):
    if v == v.p:
        return (v.d, v)
    (pd, p) = find_depth(v.p)
    d = v.d + pd
    v.d = d - p.d
    v.p = p
    return (d, p)
```

- d. Show how to implement GRAFT (r, v) , which combines the sets containing r and v , by modifying the UNION and LINK procedures. Make sure that your implementation updates pseudodistances correctly. Note that the root of a set S_i is not necessarily the root of the corresponding tree T_i .

```
def graft(r, v):
    (vd, vp) = find_depth(v)
    if r.rank <= vp.rank:
        r.d = vd + 1
        r.p = vp
        if r.rank == vp.rank:
            vp.rank += 1
    else:
        r.d = vd + 1
        vp.d = vp.d - r.d
        vp.p = r
```

- e. Give a tight bound on the worst-case running time of a sequence of m MAKE-TREE, FIND-DEPTH, and GRAFT operations, n of which are MAKE-TREE operations.

$$O(m\alpha(n))$$

21-3 Tarjan's off-line least-common-ancestors algorithm

The least common ancestor of two nodes u and v in a rooted tree T is the node w that is an ancestor of both u and v and that has the greatest depth in T . In the off-line least-common-ancestors problem, we are given a rooted tree T and an arbitrary set $P = \{\{u, v\}\}$ of unordered pairs of nodes in T , and we wish to determine the least common ancestor of each pair in P .

To solve the off-line least-common-ancestors problem, the following procedure performs a tree walk of T with the initial call $\text{LCA}(T.\text{root})$. We assume that each node is colored WHITE prior to the walk.

```

LCA(u)
1  MAKE-SET(u)
2  FIND-SET(u).ancestor = u
3  for each child v of u in T
4      LCA(v)
5      UNION(u, v)
6      FIND-SET(u).ancestor = u
7  u.color = BLACK
8  for each node v such that {u, v} \in P
9      if v.color == BLACK
10         print "The least common ancestor of"
           u "and" v "is" FIND-SET(v).ancestor

```

a. Argue that line 10 executes exactly once for each pair $\{u, v\} \in P$.

Each node is visited exactly once, if u is visited before v , then v is WHITE, line 10 will not be executed.

b. Argue that at the time of the call $\text{LCA}(u)$, the number of sets in the disjoint-set data structure equals the depth of u in T .

$\text{LCA}(v)$ increase the number of sets by 1, $\text{UNION}(u, v)$ decrease the number of sets by 1.

c. Prove that LCA correctly prints the least common ancestor of u and v for each pair $\{u, v\} \in P$.

The visited nodes always point to the current chain of search path.

d. Analyze the running time of LCA, assuming that we use the implementation of the disjoint-set data structure in Section 21.3.

$O(n^2\alpha(n))$

22 Elementary Graph Algorithms

- 22.1 Representations of graphs
- 22.2 Breadth-first search
- 22.3 Depth-first search
- 22.4 Topological sort
- 22.5 Strongly connected components
- Problems

22.1 Representations of graphs

22.1-1

Given an adjacency-list representation of a directed graph, how long does it take to compute the out-degree of every vertex? How long does it take to compute the in-degrees?

- Out-degree: $O(V + E)$
- In-degree: $O(V + E)$

22.1-2

Give an adjacency-list representation for a complete binary tree on 7 vertices. Give an equivalent adjacency-matrix representation. Assume that vertices are numbered from 1 to 7 as in a binary heap.

- Adjacency-list representation

$1 \rightarrow 2 \rightarrow 3$

$2 \rightarrow 1 \rightarrow 4 \rightarrow 5$

$3 \rightarrow 1 \rightarrow 6 \rightarrow 7$

$4 \rightarrow 2$

$5 \rightarrow 2$

$6 \rightarrow 3$

$7 \rightarrow 3$

- Adjacency-matrix representation

$$\left\{ \begin{array}{ccccccc} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{array} \right\}$$

22.1-3

The **transpose** of a directed graph $G = (V, E)$ is the graph $G^T = (V, E^T)$, where $E^T = \{(v, u) \in V \times V : (u, v) \in E\}$. Thus, G^T is G with all its edges reversed. Describe efficient algorithms for computing G^T from G , for both the adjacency-list and adjacency-matrix representations of G . Analyze the running times of your algorithms.

- Adjacency-list representation

Reconstruct, $O(V + E)$

- Adjacency-matrix representation

Transpose matrix, $O(V^2)$

22.1-4

Given an adjacency-list representation of a multigraph $G = (V, E)$, describe an $O(V + E)$ -time algorithm to compute the adjacency-list representation of the "equivalent" undirected graph $G' = (V, E')$, where E' consists of the edges in E with all multiple edges between two vertices replaced by a single edge and with all self-loops removed.

Merge sort.

22.1-5

The **square** of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$ such that $(u, v) \in E^2$ if and only G contains a path with at most two edges between u and v . Describe efficient algorithms for computing G^2 from G for both the adjacency-list and adjacency-matrix representations of G . Analyze the running times of your algorithms.

- Adjacency-list representation

```
for i in Adj[u]
    Adj^2[u].append(i)
    for j in Adj[i]
        Adj^2[u].append(j)
```

The running time depends on the distribution of edges.

- Adjacency-matrix representation

```

for i = 1 to V
    for j = 1 to V
        if a_{ij} = 1
            a^2_{ij} = 1
        else
            for k = 1 to V
                if a_{ik} == 1 and a_{kj} == 1:
                    a^2_{ij} = 1
                    break

```

$O(V^3)$

22.1-6

Most graph algorithms that take an adjacency-matrix representation as input require time $\Omega(V^2)$, but there are some exceptions. Show how to determine whether a directed graph G contains a **universal sink** - a vertex with in-degree $|V| - 1$ and out-degree 0 - in time $O(V)$, given an adjacency matrix for G .

Starting from a_{11} , if $a_{ij} = 0$ then $j = j + 1$, otherwise $i = i + 1$.

22.1-7

The **incidence matrix** of a directed graph $G = (V, E)$ with no self-loops is a $|V| \times |E|$ matrix $B = (b_{ij})$ such that

$$b_{ij} = \begin{cases} -1 & \text{if edge } j \text{ leaves vertex } i, \\ 1 & \text{if edge } j \text{ enters vertex } i, \\ 0 & \text{otherwise.} \end{cases}$$

Describe what the entries of the matrix product BB^T represent, where B^T is the transpose of B .

$(BB^T)_{ij} = \sum_{k \in E} b_{ik} \cdot b_{jk}$, the result of $b_{ik} \cdot b_{jk}$ could be 0, 1 and -1. 0 indicates i and j are not connected by edge k ; 1 indicates $i = j$; -1 indicates there is an edge from i to j or from j to i . Therefore, if $i = j$, $(BB^T)_{ij}$ is the degree of vertex i ; if $i \neq j$,

$(BB^T)_{ij}$ is the negative of number of edges connecting i and j .

22.1-8

Suppose that instead of a linked list, each array entry $Adj[u]$ is a hash table containing the vertices v for which $(u, v) \in E$. If all edge lookups are equally likely, what is the expected time to determine whether an edge is in the graph? What disadvantages does this scheme have? Suggest an alternate data structure for each edge list that solves these problems. Does your alternative have disadvantages compared to the hash table?

- Expected time: $O(1)$
- Disadvantages: More space.
- Alternative: BST, RB-Trees, ...
- Disadvantages: $O(\lg n)$

22.2 Breadth-first search

22.2-1

Show the d and π values that result from running breadth-first search on the directed graph of Figure 22.2(a), using vertex 3 as the source.

\	1	2	3	4	5	6
d	∞	3	0	2	1	1
π	NIL	4	NIL	5	3	3

22.2-2

Show the d and π values that result from running breadth-first search on the undirected graph of Figure 22.3, using vertex u as the source.

\	r	s	t	u	v	w	x	y
d	4	3	1	0	5	2	1	1
π	s	w	u	NIL	r	x	u	u

22.2-3

Show that using a single bit to store each vertex color suffices by arguing that the BFS procedure would produce the same result if lines 5 and 14 were removed.

Duplicate.

22.2-4

What is the running time of BFS if we represent its input graph by an adjacency matrix and modify the algorithm to handle this form of input?

$\Theta(V^2)$

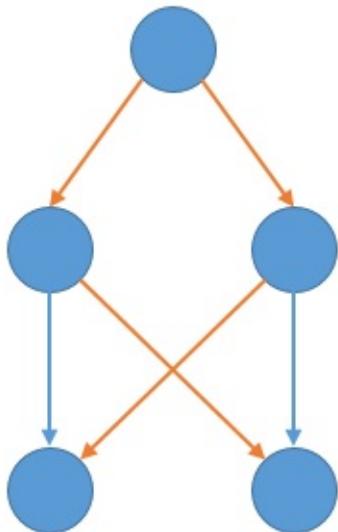
22.2-5

Argue that in a breadth-first search, the value $u.d$ assigned to a vertex u is independent of the order in which the vertices appear in each adjacency list. Using Figure 22.3 as an example, show that the breadth-first tree computed by BFS can depend on the ordering within adjacency lists.

 δ

22.2-6

Give an example of a directed graph $G = (V, E)$, a source vertex $s \in V$, and a set of tree edges $E_\pi \subseteq E$ such that for each vertex $v \in V$, the unique simple path in the graph (V, E_π) from s to v is a shortest path in G , yet the set of edges E_π cannot be produced by running BFS on G , no matter how the vertices are ordered in each adjacency list.



22.2-7

There are two types of professional wrestlers: "babyfaces" ("good guys") and "heels" ("bad guys"). Between any pair of professional wrestlers, there may or may not be a rivalry. Suppose we have n professional wrestlers and we have a list of r pairs of wrestlers for which there are rivalries. Give an $O(n + r)$ -time algorithm that determines whether it is possible to designate some of the wrestlers as babyfaces and the remainder as heels such that each rivalry is between a babyface and a heel. If it is possible to perform such a designation, your algorithm should produce it.

BFS, the new reachable node should have a different type. If the new node already have the same type with the current node, then it is impossible to perform such a designation.

22.2-8 *

The **diameter** of a tree $T = (V, E)$ is defined as $\max_{u,v \in V} \delta(u, v)$, that is, the largest of all shortest-path distances in the tree. Give an efficient algorithm to compute the diameter of a tree, and analyze the running time of your algorithm.

BFS with a random node as the source, then BFS from the node with the largest δ , the largest δ in the second BFS is the diameter of the tree, $O(V + E)$.

22.2-9

Let $G = (V, E)$ be a connected, undirected graph. Give an $O(V + E)$ -time algorithm to compute a path in G that traverses each edge in E exactly once in each direction. Describe how you can find your way out of a maze if you are given a large supply of pennies.

Eulerian path.

22.3 Depth-first search

22.3-1

Make a 3-by-3 chart with row and column labels WHITE, GRAY, and BLACK. In each cell (i, j) , indicate whether, at any point during a depth-first search of a directed graph, there can be an edge from a vertex of color i to a vertex of color j . For each possible edge, indicate what edge types it can be. Make a second such chart for depth-first search of an undirected graph.

Directed:

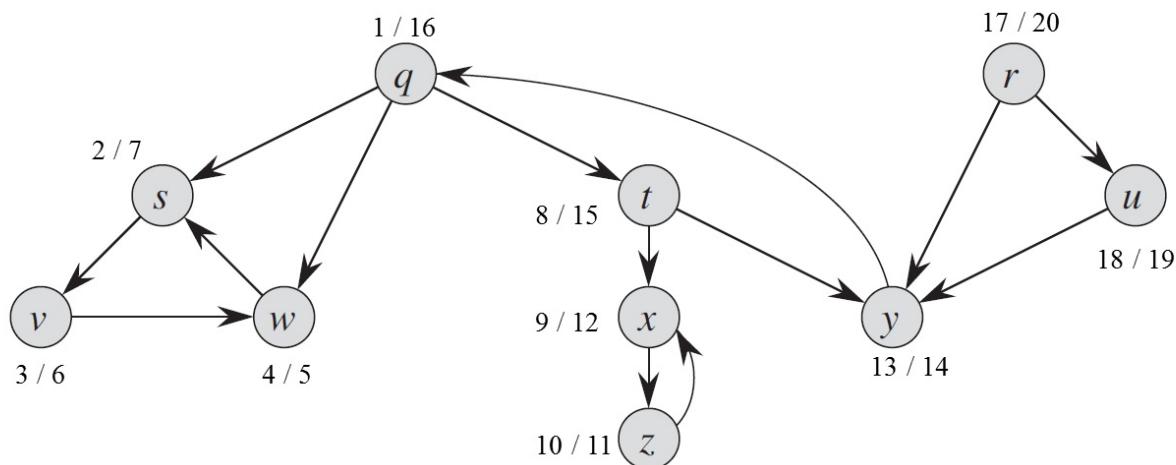
$i \setminus j$	WHITE	GRAY	BLACK
WHITE	TBFC	BC	C
GRAY	TF	TBF	TFC
BLACK		BC	TBFC

Undirected:

$i \setminus j$	WHITE	GRAY	BLACK
WHITE	TB	TB	
GRAY	TB	TB	TB
BLACK		TB	TB

22.3-2

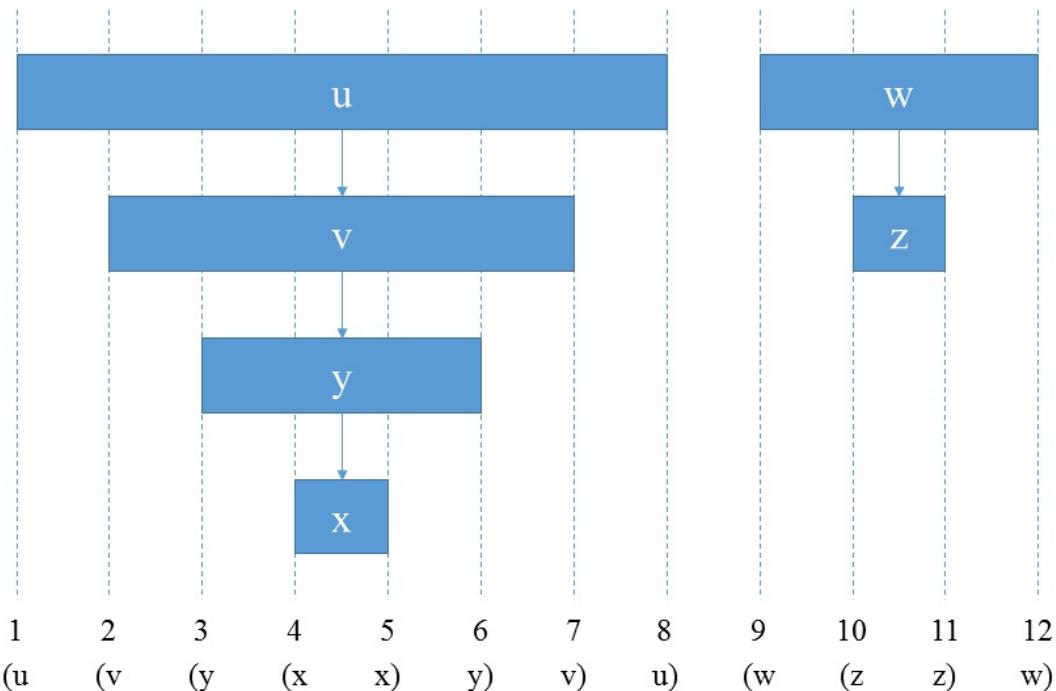
Show how depth-first search works on the graph of Figure 22.6. Assume that the **for** loop of lines 5–7 of the DFS procedure considers the vertices in alphabetical order, and assume that each adjacency list is ordered alphabetically. Show the discovery and finishing times for each vertex, and show the classification of each edge.



- Tree edges: (q, s) (s, v) (v, w) (q, t) (t, x) (x, z) (t, y) (r, u)
- Back edges: (w, s) (z, x), (y, q)
- Forward edges: (q, w)
- Cross edges: (r, y) (u, y)

22.3-3

Show the parenthesis structure of the depth-first search of Figure 22.4.



22.3-4

Show that using a single bit to store each vertex color suffices by arguing that the DFS procedure would produce the same result if line 3 of DFS-VISIT was removed.

Line 3: `color = BLACK`

22.3-5

Show that edge (u, v) is

a. a tree edge or forward edge if and only if $u.d < v.d < v.f < u.f$,

u is an ancestor of v .

b. a back edge if and only if $v.d \leq u.d < u.f \leq v.f$, and

u is a descendant of v .

c. a cross edge if and only if $v.d < v.f < u.d < u.f$.

v is visited before u .

22.3-6

Show that in an undirected graph, classifying an edge (u, v) as a tree edge or a back edge according to whether (u, v) or (v, u) is encountered first during the depth-first search is equivalent to classifying it according to the ordering of the four types in the classification scheme.

By changing an undirected graph into a directed graph with two-way edges, an equivalent result is produced.

22.3-7

Rewrite the procedure DFS, using a stack to eliminate recursion.

Goto.

22.3-8

Give a counterexample to the conjecture that if a directed graph G contains a path from u to v , and if $u.d < v.d$ in a depth-first search of G , then v is a descendant of u in the depth-first forest produced.

$$E = \{(w, u), (w, v), (u, w)\}, \text{ search } w \text{ first.}$$

22.3-9

Give a counterexample to the conjecture that if a directed graph G contains a path from u to v , then any depth-first search must result in $v.d \leq u.f$.

$$E = \{(w, u), (w, v), (u, w)\}, \text{ search } w \text{ first.}$$

22.3-10

Modify the pseudocode for depth-first search so that it prints out every edge in the directed graph G , together with its type. Show what modifications, if any, you need to make if G is undirected.

See exercises 22.3-5.

22.3-11

Explain how a vertex u of a directed graph can end up in a depth-first tree containing only u , even though u has both incoming and outgoing edges in G .

$E = \{(w, u), (u, v)\}$, search v then search u .

22.3-12

Show that we can use a depth-first search of an undirected graph G to identify the connected components of G , and that the depth-first forest contains as many trees as G has connected components. More precisely, show how to modify depth-first search so that it assigns to each vertex v an integer label $v.cc$ between 1 and k , where k is the number of connected components of G , such that $u.cc = v.cc$ if and only if u and v are in the same connected component.

```
DFS(G)
1 for each vertex u in G.V
2     u.color = WHITE
3     u.pi = NIL
4 time = 0
5 cc = 0
6 for each vertex u in G.V
7     if u.color == WHITE
8         cc = cc + 1
9         DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
1 u.cc = cc
...
```

22.3-13 *

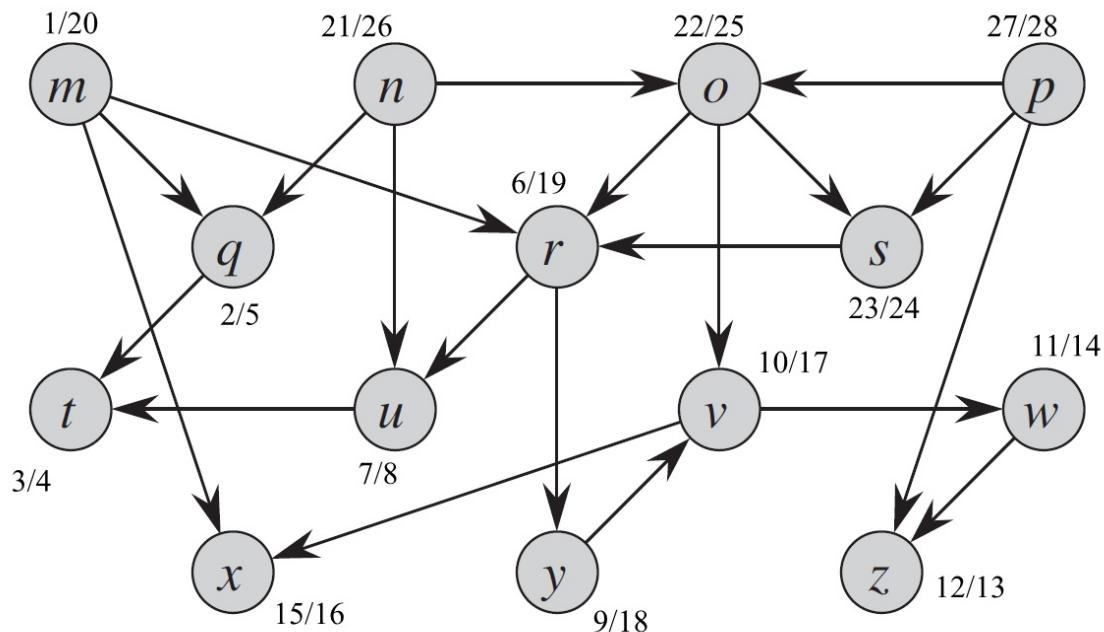
A directed graph $G = (V, E)$ is **singly connected** if $u \rightsquigarrow v$ implies that G contains at most one simple path from u to v for all vertices $u, v \in V$. Give an efficient algorithm to determine whether or not a directed graph is singly connected.

Run DFS for every vertex, if $v.color$ is BLACK, then the graph is not singly connected, $O(V \cdot (V + E))$.

22.4 Topological sort

22.4-1

Show the ordering of vertices produced by TOPOLOGICAL-SORT when it is run on the dag of Figure 22.8, under the assumption of Exercise 22.3-2.



p, n, o, s, m, r, y, v, w, z, u, q, t.

22.4-2

Give a linear-time algorithm that takes as input a directed acyclic graph $G = (V, E)$ and two vertices s and t , and returns the number of simple paths from s to t in G . For example, the directed acyclic graph of Figure 22.8 contains exactly four simple paths from vertex P to vertex v : pov , $poryv$, $posryv$, and $psryv$. (Your algorithm needs only to count the simple paths, not list them.)

Topological sort + dynamic programming.

22.4-3

Give an algorithm that determines whether or not a given undirected graph $G = (V, E)$ contains a cycle. Your algorithm should run in $O(V)$ time, independent of $|E|$.

Undirected + acyclic \rightarrow forest.

DFS, if there is a back edge, then it contains cycle. At most $|V| - 1$ edges are needed to examine since there are at most $|V| - 1$ edges in the forest.

22.4-4

Prove or disprove: If a directed graph G contains cycles, then TOPOLOGICAL-SORT (G) produces a vertex ordering that minimizes the number of "bad" edges that are inconsistent with the ordering produced.

$$E = \{(a, b), (b, c), (c, b), (c, a)\}$$

Bad edges if begins from b : $(c, b), (a, b)$. Bad edge if begins from c : (b, c) .

The number of bad edges depends on the ordering of DFS.

22.4-5

Another way to perform topological sorting on a directed acyclic graph $G = (V, E)$ is to repeatedly find a vertex of in-degree 0, output it, and remove it and all of its outgoing edges from the graph. Explain how to implement this idea so that it runs in time $O(V + E)$. What happens to this algorithm if G has cycles?

Maintain the in-degrees of the nodes. If the in-degree is 0, then add the node to a queue. When removing a node, all the nodes it connects to should subtract their in-degrees by 1.

22.5 Strongly connected components

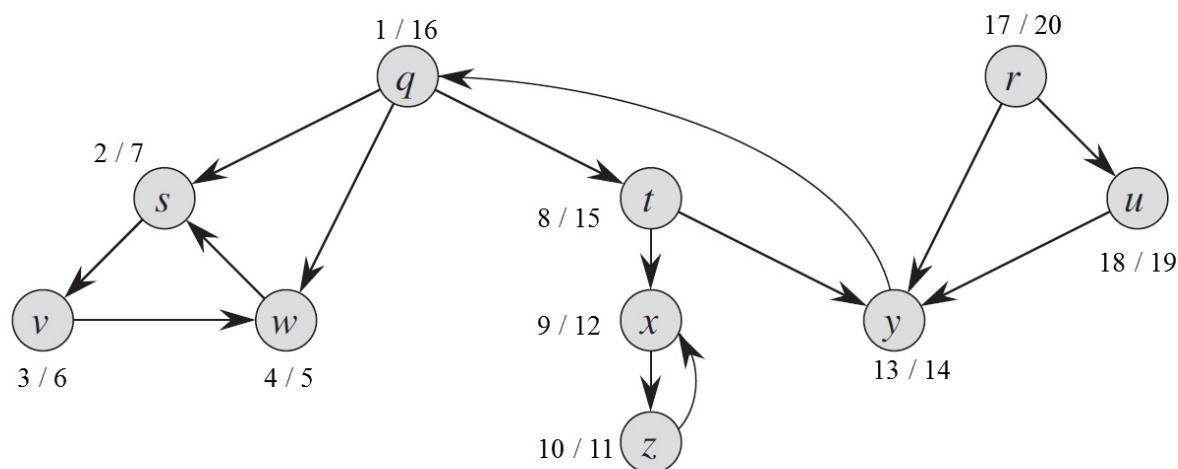
22.5-1

How can the number of strongly connected components of a graph change if a new edge is added?

$$-(K - 1) \sim +1$$

22.5-2

Show how the procedure STRONGLY-CONNECTED-COMPONENTS works on the graph of Figure 22.6. Specifically, show the finishing times computed in line 1 and the forest produced in line 3. Assume that the loop of lines 5–7 of DFS considers vertices in alphabetical order and that the adjacency lists are in alphabetical order.



$$\{r\}, \{u\}, \{q, y, t\}, \{x, z\}, \{s, w, v\}$$

22.5-3

Professor Bacon claims that the algorithm for strongly connected components would be simpler if it used the original (instead of the transpose) graph in the second depth-first search and scanned the vertices in order of *increasing* finishing times. Does this simpler algorithm always produce correct results?

No.

22.5-4

Prove that for any directed graph G , we have $((G^T)^{SCC})^T = G^{SCC}$. That is, the transpose of the component graph of G^T is the same as the component graph of G .

$$u \rightsquigarrow v \xrightarrow{T} v \rightsquigarrow u .$$

22.5-5

Give an $O(V + E)$ -time algorithm to compute the component graph of a directed graph $G = (V, E)$. Make sure that there is at most one edge between two vertices in the component graph your algorithm produces.

Add edge (u', v') if $u \in C_{u'}$, $v \in C_{v'}$ and there is an edge $(u, v) \in E$.

22.5-6

Given a directed graph $G = (V, E)$, explain how to create another graph $G' = (V, E')$ such that (a) G' has the same strongly connected components as G , (b) G' has the same component graph as G , and (c) E' is as small as possible.
Describe a fast algorithm to compute G' .

Calculate SCCs, create a loop in each SCC, connect SCCs with one edge.

22.5-7

A directed graph $G = (V, E)$ is **semiconnected** if, for all pairs of vertices $u, v \in V$, we have $u \rightsquigarrow v$ or $v \rightsquigarrow u$. Give an efficient algorithm to determine whether or not G is semiconnected. Prove that your algorithm is correct, and analyze its running time.

If $\forall i \in [1, K]$, there is an edge $(u, v) \in E$, $u \in C_i$, $v \in C_{i+1}$, then the graph is semiconnected, $O(V + E)$.

Problems

22-1 Classifying edges by breadth-first search

A depth-first forest classifies the edges of a graph into tree, back, forward, and cross edges. A breadth-first tree can also be used to classify the edges reachable from the source of the search into the same four categories.

a. Prove that in a breadth-first search of an undirected graph, the following properties hold:

1. There are no back edges and no forward edges.
2. For each tree edge (u, v) , we have $v.d = u.d + 1$.
3. For each cross edge (u, v) , we have $v.d = u.d$ or $v.d = u.d + 1$.

b. Prove that in a breadth-first search of a directed graph, the following properties hold:

1. There are no forward edges.
2. For each tree edge (u, v) , we have $v.d = u.d + 1$.
3. For each cross edge (u, v) , we have $v.d \leq u.d + 1$.
4. For each back edge (u, v) , we have $0 \leq v.d \leq u.d$.

22-2 Articulation points, bridges, and biconnected components

Let $G = (V, E)$ be a connected, undirected graph. An articulation point of G is a vertex whose removal disconnects G . A bridge of G is an edge whose removal disconnects G . A biconnected component of G is a maximal set of edges such that any two edges in the set lie on a common simple cycle. Figure 22.10 illustrates these definitions. We can determine articulation points, bridges, and biconnected components using depth-first search. Let $G_\pi = (V, E_\pi)$ be a depth-first tree of G .

a. Prove that the root of G_π is an articulation point of G if and only if it has at least two children in G_π .

At least two children => at least two components that are not connected.

b. Let v be a nonroot vertex of G_π . Prove that v is an articulation point of G if and only if v has a child s such that there is no back edge from s or any descendant of s to a proper ancestor of v .

Connect to ancestor => loop.

c. Let

$$v.\text{low} = \min \begin{cases} v.d, \\ w.d : (u, w) \text{ is a back edge for some descendant } u \text{ of } v \end{cases}$$

Show how to compute $v.\text{low}$ for all vertices $v \in V$ in $O(E)$ time.

In DFS, for each edge, $v.\text{low} = \min(v.\text{low}, w.d)$

d. Show how to compute all articulation points in $O(E)$ time.

(1) Root and have at least two children.

(2) Nonroot u and there exist an edge $(u, v) \in E$ that $v.\text{low} \geq u.d$.

e. Prove that an edge of G is a bridge if and only if it does not lie on any simple cycle of G .

No cycle => two components that are connected only by one edge.

f. Show how to compute all the bridges of G in $O(E)$ time.

$v.\text{low} > u.d$.

g. Prove that the biconnected components of G partition the nonbridge edges of G .

h. Give an $O(E)$ -time algorithm to label each edge e of G with a positive integer $e.bcc$ such that $e.bcc = e'.bcc$ if and only if e and e' are in the same biconnected component.

Delete bridges then DFS/BFS.

22-3 Euler tour

An Euler tour of a strongly connected, directed graph $G = (V, E)$ is a cycle that traverses each edge of G exactly once, although it may visit a vertex more than once.

- a. Show that G has an Euler tour if and only if $\text{in-degree}(v) = \text{out-degree}(v)$ for each vertex $v \in V$.

Part 1 : To prove Euler tour exists $\Rightarrow \text{in-degree}(v) = \text{out-degree}(v)$

Euler tour can be decomposed into a set of edge-disjoint simple cycles, that when combined form the tour. First, for each sub-cycle, since they are simple edge-disjoint cycles, each vertex v in the cycle has one edge coming into it and one edge leading out of it. Therefore, $\text{in-degree}(v) = \text{out-degree}(v)$ for each of the cycles. Second, for the entire graph, since an Euler tour exists, each simple cycle must be connected together, where each cycle has an edge coming in and an edge going out. Therefore, for each vertex v in the graph, $\text{in-degree}(v) = \text{out-degree}(v)$.

Part 2: To prove $\text{in-degree}(v) = \text{out-degree}(v) \Rightarrow$ Euler tour exists

Start from v , and chose any outgoing edge of v , say (v, u) . Since $\text{in-degree}(u) = \text{out-degree}(u)$ we can pick some outgoing edge of u and continue visiting edges. Each time we pick an edge, we can remove it from further consideration. At each vertex other than v , at the time we visit an entering edge, there must be an outgoing edge left unvisited, since $\text{in-degree} = \text{out-degree}$ for all vertices. The only vertex for which there may not be an unvisited outgoing edge is v —because we started the cycle by visiting one of v 's outgoing edges. Since there's always a leaving edge we can visit for any vertex other than v , eventually the cycle must return to v , thus proving the claim.

- b. Describe an $O(E)$ -time algorithm to find an Euler tour of G if one exists. (Hint: Merge edge-disjoint cycles.)

```

1 //defined a Vertex in a strong-connected directed graph
2 class Vertex{
3     List<Vertex> nexts;
4     List<Vertex> prevs;
5
6     static boolean reachable(Vertex v, Vertex u){
7         //return true only if exist a path from v -> u
8         //can be implemented with BFS or DFS algorithm
9     }
10
11    List<Vertex> eulerTour(){
12        List<Vertex> tour= new LinkedList<>();
13
14        for(Vertex u : this.nexts){
15            if(this.nexts.size() == 1 || reachable(u, this)){
16                tour.add(u);
17                this.nexts.remove(u);
18                tour.addAll(u.eulerTour());
19            }
20        }
21
22        return tour;
23    }
24 }

```

22-4 Reachability

Let $G = (V, E)$ be a directed graph in which each vertex $u \in V$ is labeled with a unique integer $L(U)$ from the set $\{1, 2, \dots, |V|\}$. For each vertex $u \in V$, let $R(u) = \{v \in V : u \rightsquigarrow v\}$ be the set of vertices that are reachable from u . Define $\min(u)$ to be the vertex in $R(u)$ whose label is minimum, i.e., $\min(u)$ is the vertex v such that $L(v) = \min\{L(w) : w \in R(u)\}$. Give an $O(V + E)$ -time algorithm that computes $\min(u)$ for all vertices $u \in V$.

DFS from the minimum $L(U)$ in G^T .

23 Minimum Spanning Trees

- 23.1 Growing a minimum spanning tree
- 23.2 The algorithms of Kruskal and Prim
- Problems

23.1 Growing a minimum spanning tree

23.1-1

Let (u, v) be a minimum-weight edge in a connected graph G . Show that (u, v) belongs to some minimum spanning tree of G .

$S = u$.

23.1-2

Professor Sabatier conjectures the following converse of Theorem 23.1. Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E . Let A be a subset of E that is included in some minimum spanning tree for G , let $(S, V - S)$ be any cut of G that respects A , and let (u, v) be a safe edge for A crossing $(S, V - S)$. Then, (u, v) is a light edge for the cut. Show that the professor's conjecture is incorrect by giving a counterexample.

Not light.

23.1-3

Show that if an edge (u, v) is contained in some minimum spanning tree, then it is a light edge crossing some cut of the graph.

(u, v) is a bridge in the minimum spanning tree, thus V can be divided into two components.

23.1-4

Give a simple example of a connected graph such that the set of edges $\{(u, v) : \text{there exists a cut } (S, V - S) \text{ such that } (u, v) \text{ is a light edge crossing } (S, V - S)\}$ does not form a minimum spanning tree.

$E = \{(u, v), (v, w), (w, u)\}$ with the same weight.

23.1-5

Let e be a maximum-weight edge on some cycle of connected graph $G = (V, E)$.

Prove that there is a minimum spanning tree of $G' = (V, E - \{e\})$ that is also a minimum spanning tree of G . That is, there is a minimum spanning tree of G that does not include e .

The edges in the cycle are lighter than the maximum-weight edge.

23.1-6

Show that a graph has a unique minimum spanning tree if, for every cut of the graph, there is a unique light edge crossing the cut. Show that the converse is not true by giving a counterexample.

Counterexample: $E = \{(u, v), (u, w)\}$ with the same weight.

23.1-7

Argue that if all edge weights of a graph are positive, then any subset of edges that connects all vertices and has minimum total weight must be a tree. Give an example to show that the same conclusion does not follow if we allow some weights to be nonpositive.

Not a tree: remove one edge from the cycle.

Nonpositive: $E = \{(u, v), (v, w), (w, u)\}$ with the same weight -1.

23.1-8

Let T be a minimum spanning tree of a graph G , and let L be the sorted list of the edge weights of T . Show that for any other minimum spanning tree T' of G , the list L is also the sorted list of edge weights of T' .

...

23.1-9

Let T be a minimum spanning tree of a graph $G = (V, E)$, and let V' be a subset of V . Let T' be the subgraph of T induced by V' , and let G' be the subgraph of G induced by V' . Show that if T' is connected, then T' is a minimum spanning tree of G' .

Cut V' .

23.1-10

Given a graph G and a minimum spanning tree T , suppose that we decrease the weight of one of the edges in T . Show that T is still a minimum spanning tree for G . More formally, let T be a minimum spanning tree for G with edge weights given by weight function w . Choose one edge $(x, y) \in T$ and a positive number k , and define the weight function w' by

$$w'(u, v) = \begin{cases} w(u, v) & \text{if } (u, v) \neq (x, y), \\ w(x, y) - k & \text{if } (u, v) = (x, y). \end{cases}$$

Show that T is a minimum spanning tree for G with edge weights given by w' .

Lighter.

23.1-11 *

Given a graph G and a minimum spanning tree T , suppose that we decrease the weight of one of the edges not in T . Give an algorithm for finding the minimum spanning tree in the modified graph.

If the edge (u, v) is not in T and its weight is less than some edge in the path from u to v , then replace the edge with maximum weight in the path with (u, v) .

23.2 The algorithms of Kruskal and Prim

23.2-1

Kruskal's algorithm can return different spanning trees for the same input graph G , depending on how it breaks ties when the edges are sorted into order. Show that for each minimum spanning tree T of G , there is a way to sort the edges of G in Kruskal's algorithm so that the algorithm returns T .

...

23.2-2

Suppose that we represent the graph $G = (V, E)$ as an adjacency matrix. Give a simple implementation of Prim's algorithm for this case that runs in $O(V^2)$ time.

...

23.2-3

For a sparse graph $G = (V, E)$, where $|E| = \Theta(V)$, is the implementation of Prim's algorithm with a Fibonacci heap asymptotically faster than the binary-heap implementation? What about for a dense graph, where $|E| = \Theta(V^2)$? How must the sizes $|E|$ and $|V|$ be related for the Fibonacci-heap implementation to be asymptotically faster than the binary-heap implementation?

Binary-heap: $O(E \lg V)$

Fibonacci-heap: $O(E + V \lg V)$

- $|E| = \Theta(V)$

Binary-heap: $O(V \lg V) = O(V \lg V)$

Fibonacci-heap: $O(E + V \lg V) = O(V \lg V)$

- $|E| = \Theta(V^2)$

Binary-heap: $O(V^2 \lg V) = O(V^2 \lg V)$

Fibonacci-heap: $O(V^2 + V \lg V) = O(V^2)$

23.2-4

Suppose that all edge weights in a graph are integers in the range from 1 to $|V|$. How fast can you make Kruskal's algorithm run? What if the edge weights are integers in the range from 1 to W for some constant W ?

- 1 to $|V|$

Use counting sort, $O(E\alpha(V))$.

- 1 to W

$\min(O(W + E\alpha(V)), O(E \lg V))$

23.2-5

Suppose that all edge weights in a graph are integers in the range from 1 to $|V|$. How fast can you make Prim's algorithm run? What if the edge weights are integers in the range from 1 to W for some constant W ?

- 1 to $|V|$

Use van Emde Boas trees, $O(E \lg \lg V)$.

- 1 to W

$\min(O(E \lg \lg W), O(E + V \lg V))$

23.2-6 *

Suppose that the edge weights in a graph are uniformly distributed over the halfopen interval $[0, 1)$. Which algorithm, Kruskal's or Prim's, can you make run faster?

...

23.2-7 *

Suppose that a graph G has a minimum spanning tree already computed. How quickly can we update the minimum spanning tree if we add a new vertex and incident edges to G ?

...

23.2-8

Professor Borden proposes a new divide-and-conquer algorithm for computing minimum spanning trees, which goes as follows. Given a graph $G = (V, E)$, partition the set V of vertices into two sets V_1 and V_2 such that $|V_1|$ and $|V_2|$ differ by at most 1. Let E_1 be the set of edges that are incident only on vertices in V_1 , and let E_2 be the set of edges that are incident only on vertices in V_2 . Recursively solve a minimum-spanning-tree problem on each of the two subgraphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. Finally, select the minimum-weight edge in E that crosses the cut (V_1, V_2) , and use this edge to unite the resulting two minimum spanning trees into a single spanning tree.

Either argue that the algorithm correctly computes a minimum spanning tree of G , or provide an example for which the algorithm fails.

The algorithm fails. Suppose $E = \{(u, v), (u, w), (v, w)\}$, the weight of (u, v) and (u, w) is 1, and the weight of (v, w) is 1000, partition the set into two sets $V_1 = \{u\}$ and $V_2 = \{v, w\}$.

Problems

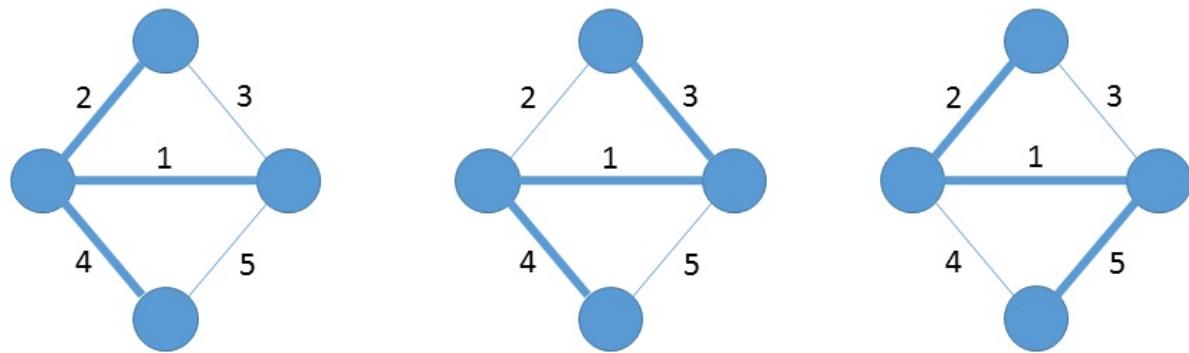
23-1 Second-best minimum spanning tree

Let $G = (V, E)$ be an undirected, connected graph whose weight function is $w : E \rightarrow \mathbb{R}$, and suppose that $|E| \geq |V|$ and all edge weights are distinct.

We define a second-best minimum spanning tree as follows. Let \mathcal{T} be the set of all spanning trees of G , and let T' be a minimum spanning tree of G . Then a **second-best minimum spanning tree** is a spanning tree T such that

$$W(T) = \min_{T'' \in \mathcal{T} - \{T'\}} \{w(T'')\}$$

- a. Show that the minimum spanning tree is unique, but that the second-best minimum spanning tree need not be unique.



- b. Let T be the minimum spanning tree of G . Prove that G contains edges $(u, v) \in T$ and $(x, y) \notin T$ such that $T - \{(u, v)\} \cup \{(x, y)\}$ is a second-best minimum spanning tree of G .

...

- c. Let T be a spanning tree of G and, for any two vertices $u, v \in V$, let $\max[u, v]$ denote an edge of maximum weight on the unique simple path between u and v in T . Describe an $O(V^2)$ -time algorithm that, given T , computes $\max[u, v]$ for all $u, v \in V$.

Search from each vertex.

- d. Give an efficient algorithm to compute the second-best minimum spanning tree of G

Find an edge $(u, v) \notin T$ with weight w that minimizes $w - \max[u, v]$. The time is $O(V^2)$.

23-2 Minimum spanning tree in sparse graphs

For a very sparse connected graph $G = (V, E)$, we can further improve upon the $O(E + V \lg V)$ running time of Prim's algorithm with Fibonacci heaps by preprocessing G to decrease the number of vertices before running Prim's algorithm. In particular, we choose, for each vertex u , the minimum-weight edge (u, v) incident on u , and we put (u, v) into the minimum spanning tree under construction. We then contract all chosen edges (see Section B.4). Rather than contracting these edges one at a time, we first identify sets of vertices that are united into the same new vertex. Then we create the graph that would have resulted from contracting these edges one at a time, but we do so by "renaming" edges according to the sets into which their endpoints were placed. Several edges from the original graph may be renamed the same as each other. In such a case, only one edge results, and its weight is the minimum of the weights of the corresponding original edges.

Initially, we set the minimum spanning tree T being constructed to be empty, and for each edge $(u, v) \in E$, we initialize the attributes $(u, v).orig = (u, v)$ and $(u, v).c = w(u, v)$. We use the *orig* attribute to reference the edge from the initial graph that is associated with an edge in the contracted graph. The *c* attribute holds the weight of an edge, and as edges are contracted, we update it according to the above scheme for choosing edge weights. The procedure MST-REDUCE takes inputs G and T , and it returns a contracted graph G' with updated attributes $orig'$ and c' . The procedure also accumulates edges of G into the minimum spanning tree T .

```

MST-REDUCE(G, T)
1 for each v in G.V
2     v.mark = FALSE
3     MAKE-SET(v)
4 for each u in G.V
5     if u.mark == FALSE
6         choose v in G.Adj[u] such that (u, v).c is minimized
7         UNION(u, v)
8         T = T [ f(u, v).origg
9         u.mark = v.mark = TRUE
10 G'.V = {FIND-SET(v) : v in G.V}
11 G'.E = empty
12 for each (x, y) in G.E
13     u = FIND-SET(x)
14     v = FIND-SET(y)
15     if (u, v) not in G'.E
16         G'.E = G'.E union {(u, v)}
17         (u, v).orig' = (x, y).orig
18         (u, v).c' = (x, y).c
19     else if (x, y).c < (u, v).c'
20         (u, v).orig' = (x, y).orig
21         (u, v).c' = (x, y).c
22 construct adjacency lists G'.Adj for G'
23 return G' and T

```

- a.** Let T be the set of edges returned by MST-REDUCE, and let A be the minimum spanning tree of the graph G' formed by the call $\text{MST-PRIM } (G', c', r)$, where c' is the weight attribute on the edges of $G'.E$ and r is any vertex in $G'.V$. Prove that $T \cup \{(x, y).orig' : (x, y) \in A\}$ is a minimum spanning tree of G .
- b.** Argue that $|G'.V| \leq |V|/2$.
- c.** Show how to implement MST-REDUCE so that it runs in $O(E)$ time. (Hint: Use simple data structures.)
- d.** Suppose that we run k phases of MST-REDUCE, using the output G' produced by one phase as the input G to the next phase and accumulating edges in T . Argue that the overall running time of the k phases is $O(kE)$.
- e.** Suppose that after running k phases of MST-REDUCE, as in part (d), we run Prim's algorithm by calling $\text{MST-PRIM } (G', c', r)$, where G' , with weight attribute c' , is returned by the last phase and r is any vertex in $G'.V$. Show how to pick k so that the overall running time is $O(E \lg \lg V)$. Argue that your choice of k minimizes the overall asymptotic running time.

f. For what values of $|E|$ (in terms of $|V|$) does Prim's algorithm with preprocessing asymptotically beat Prim's algorithm without preprocessing?

23-3 Bottleneck spanning tree

A bottleneck spanning tree T of an undirected graph G is a spanning tree of G whose largest edge weight is minimum over all spanning trees of G . We say that the value of the bottleneck spanning tree is the weight of the maximum-weight edge in T .

a. Argue that a minimum spanning tree is a bottleneck spanning tree.

Based on exercise 23.1-8, all MSTs have the same sorted weight list, thus they have the same bottleneck.

Part (a) shows that finding a bottleneck spanning tree is no harder than finding a minimum spanning tree. In the remaining parts, we will show how to find a bottleneck spanning tree in linear time.

b. Give a linear-time algorithm that given a graph G and an integer b , determines whether the value of the bottleneck spanning tree is at most b .

DFS on the graph with the edges that their weights are less or equal to b .

c. Use your algorithm for part (b) as a subroutine in a linear-time algorithm for the bottleneck-spanning-tree problem. (Hint: You may want to use a subroutine that contracts sets of edges, as in the MST-REDUCE procedure described in Problem 23-2.)

Binary search for b .

23-4 Alternative minimum-spanning-tree algorithms

In this problem, we give pseudocode for three different algorithms. Each one takes a connected graph and a weight function as input and returns a set of edges T . For each algorithm, either prove that T is a minimum spanning tree or prove that T is not a minimum spanning tree. Also describe the most efficient implementation of each algorithm, whether or not it computes a minimum spanning tree.

a.

```
MAYBE-MST-A(G, w)
1 sort the edges into nonincreasing order of edge weights w
2 T = E
3 for each edge e, taken in nonincreasing order by weight
4     if T - {e} is a connected graph
5         T = T - {e}
6 return T
```

It's a MST.

b.

```
MAYBE-MST-B(G, w)
1 T = {}
2 for each edge e, taken in arbitrary order
3     if T U {e} has no cycles
4         T = T U {e}
5 return T
```

Not.

c.

```
MAYBE-MST-C(G, w)
1 T = {}
2 for each edge e, taken in arbitrary order
3     T = T U {e}
4     if T has a cycle c
5         let e' be a maximum-weight edge on c
6         T = T - {e'}
7 return T
```

It's a MST.

24 Single-Source Shortest Paths

- 24.1 The Bellman-Ford algorithm
- 24.2 Single-source shortest paths in directed acyclic graphs
- 24.3 Dijkstra's algorithm
- 24.4 Difference constraints and shortest paths
- 24.5 Proofs of shortest-paths properties
- Problems

24.1 The Bellman-Ford algorithm

24.1-1

Run the Bellman-Ford algorithm on the directed graph of Figure 24.4, using vertex z as the source. In each pass, relax edges in the same order as in the figure, and show the d and π values after each pass. Now, change the weight of edge (z, x) to 4 and run the algorithm again, using s as the source.

\	s	t	x	y	z
d	2	4	6	9	0
π	z	x	y	s	NIL

\	s	t	x	y	z
d	0	0	2	7	-2
π	NIL	x	z	s	t

24.1-2

Prove Corollary 24.3.

No path property.

24.1-3

Given a weighted, directed graph $G = (V, E)$ with no negative-weight cycles, let m be the maximum over all vertices $v \in V$ of the minimum number of edges in a shortest path from the source s to v . (Here, the shortest path is by weight, not the number of edges.) Suggest a simple change to the Bellman-Ford algorithm that allows it to terminate in $m + 1$ passes, even if m is not known in advance.

Stop when no vertex is relaxed in a single loop.

24.1-4

Modify the Bellman-Ford algorithm so that it sets $v.d$ to $-\infty$ for all vertices v for which there is a negative-weight cycle on some path from the source to v .

```
if v.d > u.d + w(u, v)
    v.d = -inf
```

24.1-5 *

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \rightarrow \mathbb{R}$.

Give an $O(VE)$ -time algorithm to find, for each vertex $v \in V$, the value

$$\delta^*(v) = \min_{u \in V} \{\delta(u, v)\}.$$

```
RELAX(u, v, w)
1 if v.d > min(w(u, v), w(u, v) + u.d)
2     v.d = min(w(u, v), w(u, v) + u.d)
3     v.pi = u.pi
```

24.1-6 *

Suppose that a weighted, directed graph $G = (V, E)$ has a negative-weight cycle.

Give an efficient algorithm to list the vertices of one such cycle. Prove that your algorithm is correct.

Based on exercise 24.1-4, DFS from a vertex u that $u.d = -\infty$, if the weight sum on the search path is negative and the next vertex is BLACK, then the search path forms a negative-weight cycle.

24.2 Single-source shortest paths in directed acyclic graphs

24.2-1

Run DAG-SHORTEST-PATHS on the directed graph of Figure 24.5, using vertex r as the source.

\	r	s	t	x	y	z
d	0	5	3	10	7	5
π	NIL	r	r	t	t	t

24.2-2

Suppose we change line 3 of DAG-SHORTEST-PATHS to read

3 for the first $|V| - 1$ vertices, taken in topologically sorted order

Show that the procedure would remain correct.

The out-degree of the last vertex is 0.

24.2-3

The PERT chart formulation given above is somewhat unnatural. In a more natural structure, vertices would represent jobs and edges would represent sequencing constraints; that is, edge (u, v) would indicate that job u must be performed before job v . We would then assign weights to vertices, not edges. Modify the DAG-SHORTEST-PATHS procedure so that it finds a longest path in a directed acyclic graph with weighted vertices in linear time.

$$s.d = s.w, w(u, v) = v.w$$

24.2-4

Give an efficient algorithm to count the total number of paths in a directed acyclic graph. Analyze your algorithm.

$$s.num = 1, v.num = \sum_{(u,v) \in E} u.num$$

Time: $\Theta(V + E)$.

24.3 Dijkstra's algorithm

24.3-1

Run Dijkstra's algorithm on the directed graph of Figure 24.2, first using vertex s as the source and then using vertex z as the source. In the style of Figure 24.6, show the d and π values and the vertices in set S after each iteration of the **while** loop.

24.3-2

Give a simple example of a directed graph with negative-weight edges for which Dijkstra's algorithm produces incorrect answers. Why doesn't the proof of Theorem 24.6 go through when negative-weight edges are allowed?

24.3-3

Suppose we change line 4 of Dijkstra's algorithm to the following.

4 **while** $|Q| > 1$

This change causes the **while** loop to execute $|V| - 1$ times instead of $|V|$ times. Is this proposed algorithm correct?

Correct.

24.3-4

Professor Gaedel has written a program that he claims implements Dijkstra's algorithm. The program produces $v \cdot d$ and $v \cdot \pi$ for each vertex $v \in V$. Give an $O(V + E)$ -time algorithm to check the output of the professor's program. It should determine whether the d and π attributes match those of some shortest-paths tree. You may assume that all edge weights are nonnegative.

Relax on the shortest path tree.

24.3-5

Professor Newman thinks that he has worked out a simpler proof of correctness for Dijkstra's algorithm. He claims that Dijkstra's algorithm relaxes the edges of every shortest path in the graph in the order in which they appear on the path, and therefore the path-relaxation property applies to every vertex reachable from the source. Show that the professor is mistaken by constructing a directed graph for which Dijkstra's algorithm could relax the edges of a shortest path out of order.

24.3-6

We are given a directed graph $G = (V, E)$ on which each edge $(u, v) \in E$ has an associated value $r(u, v)$, which is a real number in the range $0 \leq r(u, v) \leq 1$ that represents the reliability of a communication channel from vertex u to vertex v . We interpret $r(u, v)$ as the probability that the channel from u to v will not fail, and we assume that these probabilities are independent. Give an efficient algorithm to find the most reliable path between two given vertices.

$$w(u, v) = \lg r(u, v)$$

24.3-7

Let $G = (V, E)$ be a weighted, directed graph with positive weight function $w : E \rightarrow \{1, 2, \dots, W\}$ for some positive integer W , and assume that no two vertices have the same shortest-path weights from source vertex s . Now suppose that we define an unweighted, directed graph $G' = (V \cup V', E')$ by replacing each edge $(u, v) \in E$ with $w(u, v)$ unit-weight edges in series. How many vertices does G' have? Now suppose that we run a breadth-first search on G' . Show that the order in which the breadth-first search of G' colors vertices in V black is the same as the order in which Dijkstra's algorithm extracts the vertices of V from the priority queue when it runs on G .

$$V + \sum_{(u,v) \in E} w(u, v) - E$$

24.3-8

Let $G = (V, E)$ be a weighted, directed graph with nonnegative weight function $w : E \rightarrow \{0, 1, \dots, W\}$ for some nonnegative integer W . Modify Dijkstra's algorithm to compute the shortest paths from a given source vertex s in $O(WV + E)$ time.

Use array to store vertices.

24.3-9

Modify your algorithm from Exercise 24.3-8 to run in $O((V + E) \lg W)$ time. (Hint: How many distinct shortest-path estimates can there be in $V - S$ at any point in time?)

Heap.

24.3-10

Suppose that we are given a weighted, directed graph $G = (V, E)$ in which edges that leave the source vertex s may have negative weights, all other edge weights are nonnegative, and there are no negative-weight cycles. Argue that Dijkstra's algorithm correctly finds shortest paths from s in this graph.

24.4 Difference constraints and shortest paths

24.4-1

Find a feasible solution or determine that no feasible solution exists for the following system of difference constraints:

24.4-2

Find a feasible solution or determine that no feasible solution exists for the following system of difference constraints:

No solution.

24.4-3

Can any shortest-path weight from the new vertex v_0 in a constraint graph be positive?
Explain.

24.4-4

Express the single-pair shortest-path problem as a linear program.

24.4-5

Show how to modify the Bellman-Ford algorithm slightly so that when we use it to solve a system of difference constraints with m inequalities on n unknowns, the running time is $O(nm)$.

24.4-6

Suppose that in addition to a system of difference constraints, we want to handle **equality constraints** of the form $x_i = x_j + b_k$. Show how to adapt the Bellman-Ford algorithm to solve this variety of constraint system.

24.4-7

Show how to solve a system of difference constraints by a Bellman-Ford-like algorithm that runs on a constraint graph without the extra vertex v_0 .

24.4-8 *

Let $Ax \leq b$ be a system of m difference constraints in n unknowns. Show that the Bellman-Ford algorithm, when run on the corresponding constraint graph, maximizes $\sum_{i=1}^n x_i$ subject to $Ax \leq b$ and $x_i \leq 0$ for all x_i .

24.4-9 *

Show that the Bellman-Ford algorithm, when run on the constraint graph for a system $Ax \leq b$ of difference constraints, minimizes the quantity $(\max\{x_i\} - \min\{x_i\})$ subject to $Ax \leq b$. Explain how this fact might come in handy if the algorithm is used to schedule construction jobs.

24.4-10

Suppose that every row in the matrix A of a linear program $Ax \leq b$ corresponds to a difference constraint, a single-variable constraint of the form $x_i \leq b_k$, or a single-variable constraint of the form $-x_i \leq b_k$. Show how to adapt the Bellman-Ford algorithm to solve this variety of constraint system.

24.4-11

Give an efficient algorithm to solve a system $Ax \leq b$ of difference constraints when all of the elements of b are real-valued and all of the unknowns x_i must be integers.

24.4-12 *

Give an efficient algorithm to solve a system $Ax \leq b$ of difference constraints when all of the elements of b are real-valued and a specified subset of some, but not necessarily all, of the unknowns x_i must be integers.

24.5 Proofs of shortest-paths properties

24.5-1

Give two shortest-paths trees for the directed graph of Figure 24.2 (on page 648) other than the two shown.

24.5-2

Give an example of a weighted, directed graph $G = (V, E)$ with weight function $w : E \rightarrow \mathbb{R}$ and source vertex s such that G satisfies the following property: For every edge $(u, v) \in E$, there is a shortest-paths tree rooted at s that contains (u, v) and another shortest-paths tree rooted at s that does not contain (u, v) .

24.5-3

Embellish the proof of Lemma 24.10 to handle cases in which shortest-path weights are ∞ or $-\infty$.

24.5-4

Let $G = (V, E)$ be a weighted, directed graph with source vertex s , and let G be initialized by INITIALIZE-SINGLE-SOURCE (G, s) . Prove that if a sequence of relaxation steps sets $s.\pi$ to a non-NIL value, then G contains a negative-weight cycle.

24.5-5

Let $G = (V, E)$ be a weighted, directed graph with no negative-weight edges. Let $s \in V$ be the source vertex, and suppose that we allow $v.\pi$ to be the predecessor of v on any shortest path to v from source s if $v \in V - \{s\}$ is reachable from s , and NIL otherwise. Give an example of such a graph G and an assignment of π values that produces a cycle in G_π . (By Lemma 24.16, such an assignment cannot be produced by a sequence of relaxation steps.)

24.5-6

Let $G = (V, E)$ be a weighted, directed graph with weight function $w : E \rightarrow \mathbb{R}$ and no negative-weight cycles. Let $s \in V$ be the source vertex, and let G be initialized by INITIALIZE-SINGLE-SOURCE (G, s) . Prove that for every vertex $v \in V_\pi$, there exists a path from s to v in G_π and that this property is maintained as an invariant over any sequence of relaxations.

24.5-7

Let $G = (V, E)$ be a weighted, directed graph that contains no negative-weight cycles. Let $s \in V$ be the source vertex, and let G be initialized by INITIALIZE-SINGLE-SOURCE (G, s) . Prove that there exists a sequence of $|V| - 1$ relaxation steps that produces $v.d = \delta(s, v)$ for all $v \in V$.

24.5-8

Let G be an arbitrary weighted, directed graph with a negative-weight cycle reachable from the source vertex s . Show how to construct an infinite sequence of relaxations of the edges of G such that every relaxation causes a shortest-path estimate to change.

Problems

24-1 Yen's improvement to Bellman-Ford

24-2 Nesting boxes

24-3 Arbitrage

24-4 Gabow's scaling algorithm for single-source shortest paths

24-5 Karp's minimum mean-weight cycle algorithm

24-6 Bitonic shortest paths

25 All-Pairs Shortest Paths

- 25.1 Shortest paths and matrix multiplication
- 25.2 The Floyd-Warshall algorithm
- 25.3 Johnson's algorithm for sparse graphs
- Problems

25.1 Shortest paths and matrix multiplication

25.1-1

Run SLOW-ALL-PAIRS-SHORTEST-PATHS on the weighted, directed graph of Figure 25.2, showing the matrices that result for each iteration of the loop. Then do the same for FASTER-ALL-PAIRS-SHORTEST-PATHS.

Initial:

$$\left\{ \begin{array}{cccccc} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & \infty & \infty \\ \infty & 2 & 0 & \infty & \infty & -8 \\ -4 & \infty & \infty & 0 & 3 & \infty \\ \infty & 7 & \infty & \infty & 0 & \infty \\ \infty & 5 & 10 & \infty & \infty & 0 \end{array} \right\}$$

Slow:

$m = 2$:

$$\left\{ \begin{array}{cccccc} 0 & 6 & \infty & \infty & -1 & \infty \\ -2 & 0 & \infty & 2 & 0 & \infty \\ 3 & -3 & 0 & 4 & \infty & -8 \\ -4 & 10 & \infty & 0 & -5 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & 10 & 7 & \infty & 0 \end{array} \right\}$$

$m = 3$:

$$\left\{ \begin{array}{cccccc} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -2 & -3 & 0 & -1 & 2 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 5 & 0 \end{array} \right\}$$

$m = 4$:

$$\left\{ \begin{array}{cccccc} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -3 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{array} \right\}$$

$m = 5 :$

$$\left\{ \begin{array}{cccccc} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -6 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{array} \right\}$$

Fast:

$m = 2 :$

$$\left\{ \begin{array}{cccccc} 0 & 6 & \infty & \infty & -1 & \infty \\ -2 & 0 & \infty & 2 & 0 & \infty \\ 3 & -3 & 0 & 4 & \infty & -8 \\ -4 & 10 & \infty & 0 & -5 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & 10 & 7 & \infty & 0 \end{array} \right\}$$

$m = 4 :$

$$\left\{ \begin{array}{cccccc} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -3 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{array} \right\}$$

$m = 8 :$

$$\left(\begin{array}{cccccc} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -6 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{array} \right)$$

25.1-2

Why do we require that $w_{ii} = 0$ for all $1 \leq i \leq n$?

To simplify (25.2).

25.1-3

What does the matrix

$$L^{(0)} = \left(\begin{array}{ccccc} 0 & \infty & \infty & \cdots & \infty \\ \infty & 0 & \infty & \cdots & \infty \\ \infty & \infty & 0 & \cdots & \infty \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \infty & \infty & \infty & \cdots & 0 \end{array} \right)$$

used in the shortest-paths algorithms correspond to in regular matrix multiplication?

Unit.

25.1-4

Show that matrix multiplication defined by EXTEND-SHORTEST-PATHS is associative.

25.1-5

Show how to express the single-source shortest-paths problem as a product of matrices and a vector. Describe how evaluating this product corresponds to a Bellman-Ford-like algorithm (see Section 24.1).

A vector filled with 0 except that the source is 1.

25.1-6

Suppose we also wish to compute the vertices on shortest paths in the algorithms of this section. Show how to compute the predecessor matrix Π from the completed matrix L of shortest-path weights in $O(n^3)$ time.

If $l_{ik} + w_{kj} = l_{ij}$, then $\pi_{ij} = k$.

25.1-7

We can also compute the vertices on shortest paths as we compute the shortestpath weights. Define $\pi_{ij}^{(m)}$ as the predecessor of vertex j on any minimum-weight path from i to j that contains at most m edges. Modify the EXTEND-SHORTESTPATHS and SLOW-ALL-PAIRS-SHORTEST-PATHS procedures to compute the matrices $\Pi^{(1)}, \Pi^{(2)}, \dots, \Pi^{(n-1)}$ as the matrices $L^{(1)}, L^{(2)}, \dots, L^{(n-1)}$ are computed.

If $l_{ik}^{(m-1)} + w_{kj} < l_{ij}^{(m)}$, then $\pi_{ij}^{(m)} = k$.

25.1-8

The FASTER-ALL-PAIRS-SHORTEST-PATHS procedure, as written, requires us to store $\lceil \lg(n - 1) \rceil$ matrices, each with n^2 elements, for a total space requirement of $\Theta(n^2 \lg n)$. Modify the procedure to require only $\Theta(n^2)$ space by using only two $n \times n$ matrices.

```
def fast_all_pairs_shortest_paths(w):
    n = len(w)
    m = 1
    while m < n - 1:
        w = extend_shortest_paths(w, w)
        m *= 2
    return w
```

25.1-9

Modify FASTER-ALL-PAIRS-SHORTEST-PATHS so that it can determine whether the graph contains a negative-weight cycle.

If $l_{ii} < 0$, then there is a negative-weight cycle.

25.1-10

Give an efficient algorithm to find the length (number of edges) of a minimum-length negative-weight cycle in a graph.

If $l_{ii}^{(m)} < 0$ and $l_{ii}^{(m-1)} = 0$, then the minimum-length is m .

25.2 The Floyd-Warshall algorithm

25.2-1

Run the Floyd-Warshall algorithm on the weighted, directed graph of Figure 25.2. Show the matrix $D^{(k)}$ that results for each iteration of the outer loop.

$k = 1 :$

$$\left\{ \begin{array}{cccccc} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ \infty & 2 & 0 & \infty & \infty & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ \infty & 7 & \infty & \infty & 0 & \infty \\ \infty & 5 & 10 & \infty & \infty & 0 \end{array} \right\}$$

$k = 2 :$

$$\left\{ \begin{array}{cccccc} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ 3 & 2 & 0 & 4 & 2 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & 10 & 7 & 5 & 0 \end{array} \right\}$$

$k = 3 :$

$$\left\{ \begin{array}{cccccc} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ 3 & 2 & 0 & 4 & 2 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & 10 & 7 & 5 & 0 \end{array} \right\}$$

$k = 4 :$

$$\left\{ \begin{array}{cccccc} 0 & \infty & \infty & \infty & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ 0 & 2 & 0 & 4 & -1 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{array} \right\}$$

$k = 5 :$

$$\left\{ \begin{array}{cccccc} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ 0 & 2 & 0 & 4 & -1 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{array} \right\}$$

$k = 6 :$

$$\left\{ \begin{array}{cccccc} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -6 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{array} \right\}$$

25.2-2

Show how to compute the transitive closure using the technique of Section 25.1.

25.2-3

Modify the FLOYD-WARSHALL procedure to compute the $\prod^{(k)}$ matrices according to equations (25.6) and (25.7). Prove rigorously that for all $i \in V$, the predecessor subgraph $G_{\pi,i}$ is a shortest-paths tree with root i .

25.2-4

As it appears above, the Floyd-Warshall algorithm requires $\Theta(n^3)$ space, since we compute $d_{ij}^{(k)}$ for $i, j, k = 1, 2, \dots, n$. Show that the following procedure, which simply drops all the superscripts, is correct, and thus only $\Theta(n^2)$ space is required.

25.2-5

Suppose that we modify the way in which equation (25.7) handles equality:

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} < d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \geq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$

Is this alternative definition of the predecessor matrix Π correct?

Correct.

25.2-6

How can we use the output of the Floyd-Warshall algorithm to detect the presence of a negative-weight cycle?

If $D^{(n+1)} \neq D^{(n)}$, then the graph contains negative-weight cycle.

25.2-7

Another way to reconstruct shortest paths in the Floyd-Warshall algorithm uses values $\phi_{ij}^{(k)}$ for $i, j, k = 1, 2, \dots, n$, where $\phi_{ij}^{(k)}$ is the highest-numbered intermediate vertex of a shortest path from i to j in which all intermediate vertices are in the set $\{1, 2, \dots, k\}$. Give a recursive formulation for $\phi_{ij}^{(k)}$, modify the FLOYD-WARSHALL procedure to compute the $\phi_{ij}^{(k)}$ values, and rewrite the PRINT-ALLPAIRS-SHORTEST-PATH procedure to take the matrix $\Phi = (\phi_{ij}^{(n)})$ as an input. How is the matrix Φ like the s table in the matrix-chain multiplication problem of Section 15.2?

25.2-8

Give an $O(VE)$ -time algorithm for computing the transitive closure of a directed graph $G = (V, E)$.

DFS from each vertex.

25.2-9

Suppose that we can compute the transitive closure of a directed acyclic graph in $f(|V|, |E|)$ time, where f is a monotonically increasing function of $|V|$ and $|E|$.

Show that the time to compute the transitive closure $G^* = (V, E^*)$ of a general directed graph $G = (V, E)$ is then $f(|V|, |E|) + O(V + E^*)$.

All the pairs of vertices in one SCC are connected, and the SCCs forms a directed acyclic graph.

25.3 Johnson's algorithm for sparse graphs

25.3-1

Use Johnson's algorithm to find the shortest paths between all pairs of vertices in the graph of Figure 25.2. Show the values of h and \hat{w} computed by the algorithm.

25.3-2

What is the purpose of adding the new vertex s to V' , yielding V' ?

To reach all the vertices.

25.3-3

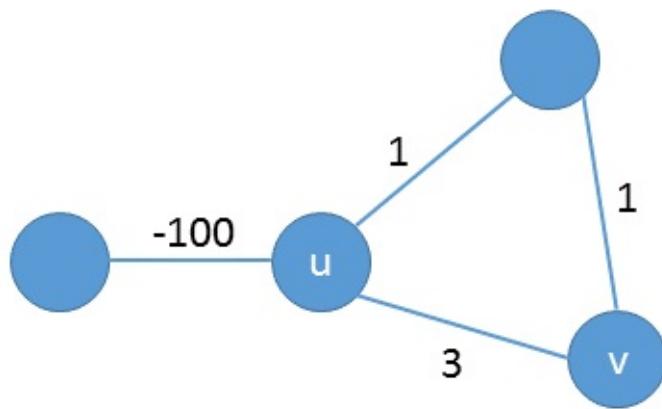
Suppose that $w(u, v) \geq 0$ for all edges $(u, v) \in E$. What is the relationship between the weight functions w and \hat{w} ?

$$h(u) = h(v) = 0, w = \hat{w}$$

25.3-4

Professor Greenstreet claims that there is a simpler way to reweight edges than the method used in Johnson's algorithm. Letting $w^* = \min_{(u,v) \in E} \{w(u, v)\}$, just define $\hat{w}(u, v) = w(u, v) - w^*$ for all edges $(u, v) \in E$. What is wrong with the professor's method of reweighting?

$$\hat{w}(p) = w(p) - (k - 1)w^*$$

**25.3-5**

Suppose that we run Johnson's algorithm on a directed graph G with weight function w . Show that if G contains a 0-weight cycle c , then $\hat{w}(u, v) = 0$ for every edge (u, v) in c .

$\delta(s, v) - \delta(s, u) \leq w(u, v)$, if $\delta(s, v) - \delta(s, u) < w(u, v)$, then we have
 $\delta(s, u) \leq \delta(s, v) + (0 - w(u, v)) < \delta(s, u) + w(u, v) - w(u, v) = \delta(s, u)$,
which is impossible, thus $\delta(s, v) - \delta(s, u) = w(u, v)$,
 $\hat{w}(u, v) = w(u, v) + \delta(s, u) - \delta(s, v) = 0$.

25.3-6

Professor Michener claims that there is no need to create a new source vertex in line 1 of JOHNSON. He claims that instead we can just use $G' = G$ and let s be any vertex. Give an example of a weighted, directed graph G for which incorporating the professor's idea into JOHNSON causes incorrect answers. Then show that if G is strongly connected (every vertex is reachable from every other vertex), the results returned by JOHNSON with the professor's modification are correct.

$$E = \{(u, s)\}$$

Problems

25-1 Transitive closure of a dynamic graph

Suppose that we wish to maintain the transitive closure of a directed graph $G = (V, E)$ as we insert edges into E . That is, after each edge has been inserted, we want to update the transitive closure of the edges inserted so far. Assume that the graph G has no edges initially and that we represent the transitive closure as a boolean matrix.

- a. Show how to update the transitive closure $G^* = (V, E^*)$ of a graph $G = (V, E)$ in $O(V^2)$ time when a new edge is added to G .

Suppose the inverted edge is (u, v) , then if (a, u) is true and (v, b) is true, then (a, b) is true.

- b. Give an example of a graph G and an edge e such that $\Omega(V^2)$ time is required to update the transitive closure after the insertion of e into G , no matter what algorithm is used.

Two connected components.

- c. Describe an efficient algorithm for updating the transitive closure as edges are inserted into the graph. For any sequence of n insertions, your algorithm should run in total time $\sum_{i=1}^n t_i = O(V^3)$, where t_i is the time to update the transitive closure upon inserting the i th edge. Prove that your algorithm attains this time bound.

If (a, u) is true, (a, v) is not true and (v, b) is true, then (a, b) is true.

25-2 Shortest paths in ϵ -dense graphs

A graph $G = (V, E)$ is ϵ -dense if $|E| = \Theta(V^{1+\epsilon})$ for some constant ϵ in the range $0 < \epsilon \leq 1$. By using d -ary min-heaps (see Problem 6-2) in shortest-paths algorithms on ϵ -dense graphs, we can match the running times of Fibonacci-heap-based algorithms without using as complicated a data structure.

a. What are the asymptotic running times for INSERT, EXTRACT-MIN, and DECREASE-KEY, as a function of d and the number n of elements in a d -ary min-heap? What are these running times if we choose $d = \Theta(n^\alpha)$ for some constant $0 < \alpha \leq 1$? Compare these running times to the amortized costs of these operations for a Fibonacci heap.

- **INSERT:** $\Theta(\log_d n) = \Theta(1/\alpha)$
- **EXTRACT-MIN:** $\Theta(d \log_d n) = \Theta(n^\alpha/\alpha)$
- **DECREASE-KEY:** $\Theta(\log_d n) = \Theta(1/\alpha)$

b. Show how to compute shortest paths from a single source on an ϵ -dense directed graph $G = (V, E)$ with no negative-weight edges in $O(E)$ time. (Hint: Pick d as a function of ϵ .)

Dijkstra, $O(d \log_d V \cdot V + \log_d V \cdot E)$, if $d = V^\epsilon$, then

$$\begin{aligned}
 & O(d \log_d V \cdot V + \log_d V \cdot E) \\
 &= O(V^\epsilon \cdot V/\epsilon + E/\epsilon) \\
 &= O((V^{1+\epsilon} + E)/\epsilon) \\
 &= O((E + E)/\epsilon) \\
 &= O(E)
 \end{aligned}$$

c. Show how to solve the all-pairs shortest-paths problem on an ϵ -dense directed graph $G = (V, E)$ with no negative-weight edges in $O(VE)$ time.

Run $|V|$ times Dijkstra, since the algorithm is $O(E)$ based on b , the total time is $O(VE)$.

d. Show how to solve the all-pairs shortest-paths problem in $O(VE)$ time on an ϵ -dense directed graph $G = (V, E)$ that may have negative-weight edges but has no negative-weight cycles.

Johnson's reweight is $O(VE)$.

26 Maximum Flow

- 26.1 Flow networks
- 26.2 The Ford-Fulkerson method
- 26.3 Maximum bipartite matching
- 26.4 Push-relabel algorithms
- 26.5 The relabel-to-front algorithm
- Problems

26.1 Flow networks

26.1-1

Show that splitting an edge in a flow network yields an equivalent network. More formally, suppose that flow network G contains edge (u, v) , and we create a new flow network G' by creating a new vertex x and replacing (u, v) by new edges (u, x) and (x, v) with $c(u, x) = c(x, v) = c(u, v)$. Show that a maximum flow in G' has the same value as a maximum flow in G .

$$f(u, x) = f(x, v)$$

26.1-2

Extend the flow properties and definitions to the multiple-source, multiple-sink problem. Show that any flow in a multiple-source, multiple-sink flow network corresponds to a flow of identical value in the single-source, single-sink network obtained by adding a supersource and a supersink, and vice versa.

Capacity constraint: for all $u, v \in V$, we require $0 \leq f(u, v) \leq c(u, v)$.

Flow conservation: for all $u \in V - S - T$, we require

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$$

26.1-3

Suppose that a flow network $G = (V, E)$ violates the assumption that the network contains a path $s \rightsquigarrow v \rightsquigarrow t$ for all vertices $v \in V$. Let u be a vertex for which there is no path $s \rightsquigarrow u \rightsquigarrow t$. Show that there must exist a maximum flow f in G such that $f(u, v) = f(v, u) = 0$ for all vertices $v \in V$.

Cannot flow in or flow out.

26.1-4

Let f be a flow in a network, and let α be a real number. The **scalar flow product**, denoted αf , is a function from $V \times V$ to \mathbb{R} defined by

$$(\alpha f)(u, v) = \alpha \cdot f(u, v)$$

Prove that the flows in a network form a **convex set**. That is, show that if f_1 and f_2 are flows, then so is $\alpha f_1 + (1 - \alpha) f_2$ for all α in the range $0 \leq \alpha \leq 1$.

26.1-5

State the maximum-flow problem as a linear-programming problem.

$$\begin{aligned} \max \quad & \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) \\ \text{s. t.} \quad & 0 \leq f(u, v) \leq c(u, v) \\ & \sum_{v \in V} f(v, u) - \sum_{v \in V} f(u, v) = 0 \end{aligned}$$

26.1-6

Professor Adam has two children who, unfortunately, dislike each other. The problem is so severe that not only do they refuse to walk to school together, but in fact each one refuses to walk on any block that the other child has stepped on that day. The children have no problem with their paths crossing at a corner. Fortunately both the professor's house and the school are on corners, but beyond that he is not sure if it is going to be possible to send both of his children to the same school. The professor has a map of his town. Show how to formulate the problem of determining whether both his children can go to the same school as a maximum-flow problem.

The capacity of each path is 1, the maximum-flow should be greater than 1.

26.1-7

Suppose that, in addition to edge capacities, a flow network has **vertex capacities**.

That is each vertex v has a limit $l(v)$ on how much flow can pass through v . Show how to transform a flow network $G = (V, E)$ with vertex capacities into an equivalent flow network $G' = (V', E')$ without vertex capacities, such that a maximum flow in G' has the same value as a maximum flow in G . How many vertices and edges does G' have?

For each vertex v , transform it to an edge (v, v') with capacity $l(v)$. G' has $2V$ vertices and $V + E$ edges.

26.2 The Ford-Fulkerson method

26.2-1

Prove that the summations in equation (26.6) equal the summations in equation (26.7).

$$\sum_{v \in V_2} f(s, v) = 0, \sum_{v \in V_1} f(v, s)$$

26.2-2

In Figure 26.1(b), what is the flow across the cut $(\{s, v_2, v_4\}, \{v_1, v_3, t\})$? What is the capacity of this cut?

$$f(S, T) = f(s, v_1) + f(v_2, v_1) + f(v_4, v_3) + f(v_4, t) - f(v_3, v_2) = 11 + 1 + 7 + 4 - 4 = 19$$

$$c(S, T) = c(s, v_1) + c(v_2, v_1) + c(v_4, v_3) + c(v_4, t) = 16 + 4 + 7 + 4 = 31$$

26.2-3

Show the execution of the Edmonds-Karp algorithm on the flow network of Figure 26.1(a).

26.2-4

In the example of Figure 26.6, what is the minimum cut corresponding to the maximum flow shown? Of the augmenting paths appearing in the example, which one cancels flow?

26.2-5

Recall that the construction in Section 26.1 that converts a flow network with multiple sources and sinks into a single-source, single-sink network adds edges with infinite capacity. Prove that any flow in the resulting network has a finite value if the edges of the original network with multiple sources and sinks have finite capacity.

Flow in equals flow out.

26.2-6

Suppose that each source s_i in a flow network with multiple sources and sinks produces exactly p_i units of flow, so that $\sum_{v \in V} f(s_i, v) = p_i$. Suppose also that each sink t_j consumes exactly q_j units, so that $\sum_{v \in V} f(v, t_j) = q_j$, where $\sum_i p_i = \sum_j q_j$. Show how to convert the problem of finding a flow f that obeys these additional constraints into the problem of finding a maximum flow in a single-source, single-sink flow network.

$$c(s, s_i) = p_i, c(t_j, t) = q_j.$$

26.2-7

Prove Lemma 26.2.

26.2-8

Suppose that we redefine the residual network to disallow edges into s . Argue that the procedure FORD-FULKERSON still correctly computes a maximum flow.

Correct.

26.2-9

Suppose that both f and f' are flows in a network G and we compute flow $f \uparrow f'$. Does the augmented flow satisfy the flow conservation property? Does it satisfy the capacity constraint?

It satisfies the flow conservation property and doesn't satisfy the capacity constraint.

26.2-10

Show how to find a maximum flow in a network $G = (V, E)$ by a sequence of at most $|E|$ augmenting paths. (Hint: Determine the paths after finding the maximum flow.)

Find the minimum cut.

26.2-11

The **edge connectivity** of an undirected graph is the minimum number k of edges that must be removed to disconnect the graph. For example, the edge connectivity of a tree is 1, and the edge connectivity of a cyclic chain of vertices is 2. Show how to determine the edge connectivity of an undirected graph $G = (V, E)$ by running a maximum-flow algorithm on at most $|V|$ flow networks, each having $O(V)$ vertices and $O(E)$ edges.

Use each v as the source, find the minimum minimum cut.

26.2-12

Suppose that you are given a flow network G , and G has edges entering the source s . Let f be a flow in G in which one of the edges (v, s) entering the source has $f(v, s) = 1$. Prove that there must exist another flow f' with $f'(v, s) = 0$ such that $|f| = |f'|$. Give an $O(E)$ -time algorithm to compute f' , given f , and assuming that all edge capacities are integers.

26.2-13

Suppose that you wish to find, among all minimum cuts in a flow network G with integral capacities, one that contains the smallest number of edges. Show how to modify the capacities of G to create a new flow network G' in which any minimum cut in G' is a minimum cut with the smallest number of edges in G .

26.3 Maximum bipartite matching

26.3-1

Run the Ford-Fulkerson algorithm on the flow network in Figure 26.8(c) and show the residual network after each flow augmentation. Number the vertices in L top to bottom from 1 to 5 and in R top to bottom from 6 to 9. For each iteration, pick the augmenting path that is lexicographically smallest.

26.3-2

Prove Theorem 26.10.

26.3-3

Let $G = (V, E)$ be a bipartite graph with vertex partition $V = L \cup R$, and let G' be its corresponding flow network. Give a good upper bound on the length of any augmenting path found in G' during the execution of FORD-FULKERSON.

26.3-4 *

A **perfect matching** is a matching in which every vertex is matched. Let $G = (V, E)$ be an undirected bipartite graph with vertex partition $V = L \cup R$, where $|L| = |R|$. For any $X \subseteq V$, define the **neighborhood** of X as

$$N(X) = \{y \in V : (x, y) \in E \text{ for some } x \in X\},$$

that is, the set of vertices adjacent to some member of X . Prove **Hall's theorem**: there exists a perfect matching in G if and only if $|A| \leq |N(A)|$ for every subset $A \subseteq L$.

26.3-5 *

We say that a bipartite graph $G = (V, E)$, where $V = L \cup R$, is **d -regular** if every vertex $v \in V$ has degree exactly d . Every d -regular bipartite graph has $|L| = |R|$. Prove that every d -regular bipartite graph has a matching of cardinality $|L|$ by arguing that a minimum cut of the corresponding flow network has capacity $|L|$.

26.4 Push-relabel algorithms

26.4-1

Prove that, after the procedure INITIALIZE-PREFLOW (G, S) terminates, we have $s \cdot e \leq -|f^*|$, where f^* is a maximum flow for G .

26.4-2

Show how to implement the generic push-relabel algorithm using $O(V)$ time per relabel operation, $O(1)$ time per push, and $O(1)$ time to select an applicable operation, for a total time of $O(V^2 E)$.

26.4-3

Prove that the generic push-relabel algorithm spends a total of only $O(VE)$ time in performing all the $O(V^2)$ relabel operations.

26.4-4

Suppose that we have found a maximum flow in a flow network $G = (V, E)$ using a push-relabel algorithm. Give a fast algorithm to find a minimum cut in G .

26.4-5

Give an efficient push-relabel algorithm to find a maximum matching in a bipartite graph. Analyze your algorithm.

26.4-6

Suppose that all edge capacities in a flow network $G = (V, E)$ are in the set $\{1, 2, \dots, k\}$. Analyze the running time of the generic push-relabel algorithm in terms of $|V|$, $|E|$, and k . (Hint: How many times can each edge support a nonsaturating push before it becomes saturated?)

26.4-7

Show that we could change line 6 of INITIALIZE-PREFLOW to

6 s.h = |G.V| - 2

without affecting the correctness or asymptotic performance of the generic pushrelabel algorithm.

26.4-8

Let $\delta_f(u, v)$ be the distance (number of edges) from u to v in the residual network G_f . Show that the GENERIC-PUSH-RELABEL procedure maintains the properties that $u.h < |V|$ implies $u.h \leq \delta_f(u, t)$ and that $u.h \geq |V|$ implies $u.h - |V| \leq \delta_f(u, s)$.

26.4-9 *

As in the previous exercise, let $\delta_f(u, v)$ be the distance from u to v in the residual network G_f . Show how to modify the generic push-relabel algorithm to maintain the property that $u.h < |V|$ implies $u.h = \delta_f(u, t)$ and that $u.h \geq |V|$ implies $u.h - |V| = \delta_f(u, s)$. The total time that your implementation dedicates to maintaining this property should be $O(VE)$.

26.4-10

Show that the number of nonsaturating pushes executed by the GENERIC-PUSH-RELABEL procedure on a flow network $G = (V, E)$ is at most $4|V|^2|E|$ for $|V| \geq 4$.

26.5 The relabel-to-front algorithm

26.5-1

Illustrate the execution of RELABEL-TO-FRONT in the manner of Figure 26.10 for the flow network in Figure 26.1(a). Assume that the initial ordering of vertices in L is $\langle v_1, v_2, v_3, v_4 \rangle$ and that the neighbor lists are

$$v_1.N = \langle s, v_2, v_3 \rangle, v_2.N = \langle s, v_1, v_3, v_4 \rangle, v_3.N = \langle v_1, v_2, v_4, t \rangle,$$
$$v_4.N = \langle v_2, v_3, t \rangle,$$

26.5-2 *

We would like to implement a push-relabel algorithm in which we maintain a first-in, first-out queue of overflowing vertices. The algorithm repeatedly discharges the vertex at the head of the queue, and any vertices that were not overflowing before the discharge but are overflowing afterward are placed at the end of the queue. After the vertex at the head of the queue is discharged, it is removed. When the queue is empty, the algorithm terminates. Show how to implement this algorithm to compute a maximum flow in $O(V^3)$ time.

26.5-3

Show that the generic algorithm still works if RELABEL updates $u.h$ by simply computing $u.h = u.h + 1$. How would this change affect the analysis of RELABEL-TO-FRONT?

26.5-4 *

Show that if we always discharge a highest overflowing vertex, we can make the push-relabel method run in $O(V^3)$ time.

26.5-5

Suppose that at some point in the execution of a push-relabel algorithm, there exists an integer $0 < k \leq |V| - 1$ for which no vertex has $v.h = k$. Show that all vertices with $v.h > k$ are on the source side of a minimum cut. If such a k exists, the **gap heuristic** updates every vertex $v \in V - \{s\}$ for which $v.h > k$, to set $v.h = \max(v.h, |V| + 1)$. Show that the resulting attribute h is a height function. (The gap heuristic is crucial in making implementations of the push-relabel method perform well in practice.)

Problems

26-1 Escape problem

26-2 Minimum path cover

26-3 Algorithmic consulting

26-4 Updating maximum flow

26-5 Maximum flow by scaling

26-6 The Hopcroft-Karp bipartite matching algorithm

27 Multithreaded Algorithms

- 27.1 The basics of dynamic multithreading
- 27.2 Multithreaded matrix multiplication
- 27.3 Multithreaded merge sort
- Problems

27.1 The basics of dynamic multithreading

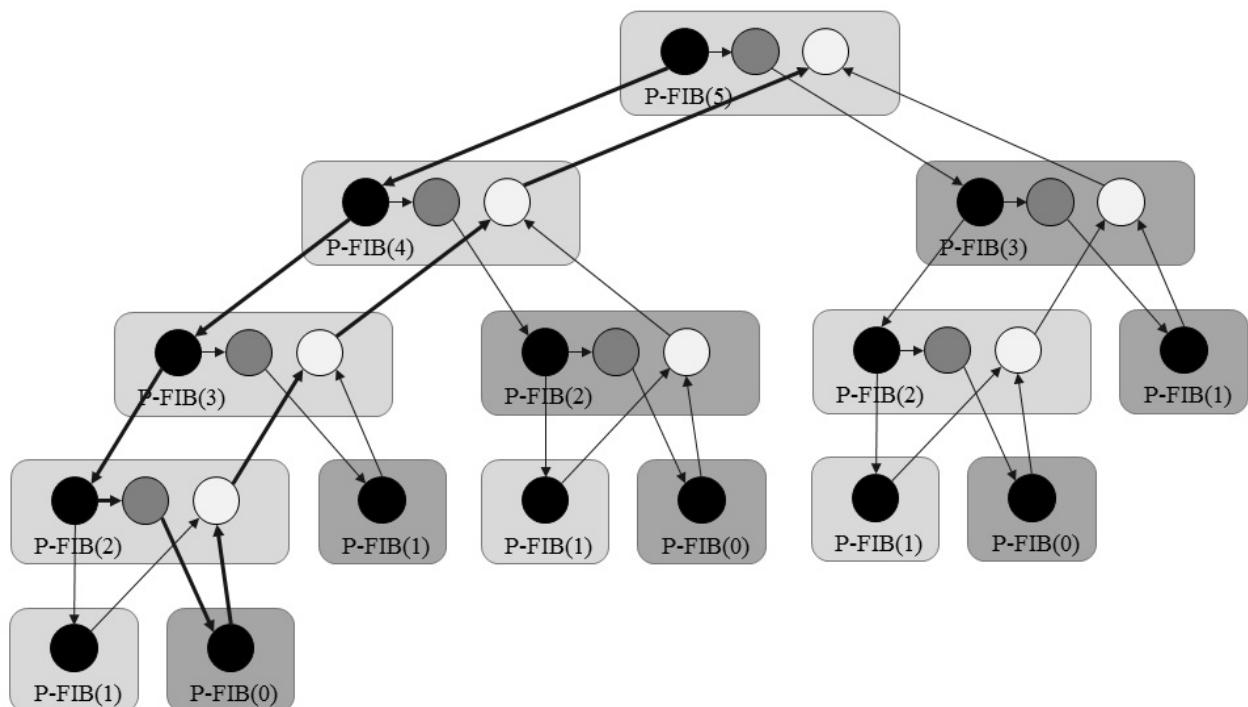
27.1-1

Suppose that we spawn P-FIB^(n - 2) in line 4 of P-FIB, rather than calling it as is done in the code. What is the impact on the asymptotic work, span, and parallelism?

No change.

27.1-2

Draw the computation dag that results from executing P-FIB(5). Assuming that each strand in the computation takes unit time, what are the work, span, and parallelism of the computation? Show how to schedule the dag on 3 processors using greedy scheduling by labeling each strand with the time step in which it is executed.



- Work: $T_1 = 29$
- Span: $T_\infty = 9$

- Parallelism: $T_1/T_\infty \approx 3.2$

27.1-3

Prove that a greedy scheduler achieves the following time bound, which is slightly stronger than the bound proven in Theorem 27.1:

$$T_P \leq \frac{T_1 - T_\infty}{P} + T_\infty$$

$T_1 - T_\infty$ is the number of strands that are not belong to the longest path.

27.1-4

Construct a computation dag for which one execution of a greedy scheduler can take nearly twice the time of another execution of a greedy scheduler on the same number of processors. Describe how the two executions would proceed.

The critical path is twice the length of the other path.

27.1-5

Professor Karan measures her deterministic multithreaded algorithm on 4, 10, and 64 processors of an ideal parallel computer using a greedy scheduler. She claims that the three runs yielded $T_4 = 80$ seconds, $T_{10} = 42$ seconds, and $T_{64} = 10$ seconds.

Argue that the professor is either lying or incompetent. (Hint: Use the work law (27.2), the span law (27.3), and inequality (27.5) from Exercise 27.1-3.)

Based on span law:

$$T_\infty \leq T_P = \min\{80, 42, 10\} = 10$$

Based on inequality (27.5):

$$\begin{cases} T_1 + 3T_\infty & \geq 320 \\ T_1 + 9T_\infty & \geq 420 \end{cases}$$

$6T_\infty \geq 100$, $T_\infty \geq 16$, which contradicts the span law.

27.1-6

Give a multithreaded algorithm to multiply an $n \times n$ matrix by an n -vector that achieves $\Theta(n^2/\lg n)$ parallelism while maintaining $\Theta(n^2)$ work.

```

VEC-TIMES-VEC(a, b, l, r)
1 if l == r
2     return a[l] * b[r]
2 m = floor((l + r) / 2)
3 spawn sum_l = VEC-TIMES-VEC(a, b, l, m)
4 spawn sum_r = VEC-TIMES-VEC(a, b, m + 1, r)
5 sync
6 return sum_l + sum_r

```

The multiply of two vectors is thus $\Theta(\lg n)$, there are n vectors to multiply simultaneously, and the outer parallel for is optimized to $\Theta(\lg n)$, therefore $T_\infty = \Theta(\lg n) + \Theta(\lg n) = \Theta(\lg n)$, since $T_1 = \Theta(n^2)$, then the parallelism $T_1/T_\infty = \Theta(n^2/\lg n)$.

27.1-7

Consider the following multithreaded pseudocode for transposing an $n \times n$ matrix A in place:

```

P-TRANPOSE(A)
1 n = A.rows
2 parallel for j = 2 to n
3     parallel for i = 1 to j - 1
4         exchange a_ij with a_ji

```

Analyze the work, span, and parallelism of this algorithm.

- **Work:** $T_1 = \Theta(n^2)$
- **Span:** $T_\infty = \Theta(\lg n) + \Theta(\lg n) + \Theta(1) = \Theta(\lg n)$
- **Parallelism:** $T_1/T_\infty = \Theta(n^2/\lg n)$

27.1-8

Suppose that we replace the **parallel for** loop in line 3 of P-TRANPOSE (see Exercise 27.1-7) with an ordinary **for** loop. Analyze the work, span, and parallelism of the resulting algorithm.

- **Work:** $T_1 = \Theta(n^2)$
- **Span:** $T_\infty = \Theta(\lg n) + \Theta(n) = \Theta(n)$
- **Parallelism:** $T_1/T_\infty = \Theta(n)$

27.1-9

For how many processors do the two versions of the chess programs run equally fast, assuming that $T_P = T_1/P + T_\infty$?

$$\begin{aligned}T_1/P + T_\infty &= T'_1/P + T'_\infty \\2048 + P &= 1024 + 8P \\P &= 1024/7 \\&\approx 146\end{aligned}$$

27.2 Multithreaded matrix multiplication

27.2-1

Draw the computation dag for computing P-SQUARE-MATRIX-MULTIPLY on 2×2 matrices, labeling how the vertices in your diagram correspond to strands in the execution of the algorithm. Use the convention that spawn and call edges point downward, continuation edges point horizontally to the right, and return edges point upward. Assuming that each strand takes unit time, analyze the work, span, and parallelism of this computation.

27.2-2

Repeat Exercise 27.2-1 for P-MATRIX-MULTIPLY-RECURSIVE.

27.2-3

Give pseudocode for a multithreaded algorithm that multiplies two $n \times n$ matrices with work $\Theta(n^3)$ but span only $\Theta(\lg n)$. Analyze your algorithm.

Based on exercise 27.1-6, the product of two vectors is $\Theta(\lg n)$, thus
 $T_\infty = \Theta(\lg n) + \Theta(\lg n) + \Theta(\lg n) = \Theta(\lg n)$.

27.2-4

Give pseudocode for an efficient multithreaded algorithm that multiplies a $p \times q$ matrix by a $q \times r$ matrix. Your algorithm should be highly parallel even if any of p , q , and r are 1. Analyze your algorithm.

$$T_\infty = \Theta(\min\{\lg p, \lg q, \lg r\})$$

27.2-5

Give pseudocode for an efficient multithreaded algorithm that transposes an $n \times n$ matrix in place by using divide-and-conquer to divide the matrix recursively into four $n/2 \times n/2$ submatrices. Analyze your algorithm.

$$M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}, M^T = \begin{pmatrix} A^T & C^T \\ B^T & D^T \end{pmatrix}, T_\infty(n) = T_\infty(n/2) + \Theta(1),$$
$$T_\infty = \Theta(\lg n)$$

27.2-6

Give pseudocode for an efficient multithreaded implementation of the Floyd-Warshall algorithm (see Section 25.2), which computes shortest paths between all pairs of vertices in an edge-weighted graph. Analyze your algorithm.

i and j can be paralleled, $T_\infty = \Theta(n \lg n)$.

27.3 Multithreaded merge sort

27.3-1

Explain how to coarsen the base case of P-MERGE.

27.3-2

Instead of finding a median element in the larger subarray, as P-MERGE does, consider a variant that finds a median element of all the elements in the two sorted subarrays using the result of Exercise 9.3-8. Give pseudocode for an efficient multithreaded merging procedure that uses this median-finding procedure. Analyze your algorithm.

27.3-3

Give an efficient multithreaded algorithm for partitioning an array around a pivot, as is done by the PARTITION procedure on page 171. You need not partition the array in place. Make your algorithm as parallel as possible. Analyze your algorithm. (Hint: You may need an auxiliary array and may need to make more than one pass over the input elements.)

Parallel for then merge.

27.3-4

Give a multithreaded version of RECURSIVE-FFT on page 911. Make your implementation as parallel as possible. Analyze your algorithm.

27.3-5 *

Give a multithreaded version of RANDOMIZED-SELECT on page 216. Make your implementation as parallel as possible. Analyze your algorithm. (Hint: Use the partitioning algorithm from Exercise 27.3-3.)

$$T_{\infty} = \Theta(\lg^2 n)$$

27.3-6 *

Show how to multithread SELECT from Section 9.3. Make your implementation as parallel as possible. Analyze your algorithm.

$$T_{\infty} = \Theta(\lg^2 n)$$

Problems

27-1 Implementing parallel loops using nested parallelism

Consider the following multithreaded algorithm for performing pairwise addition on n -element arrays $A[1 \dots n]$ and $B[1 \dots n]$, storing the sums in $C[1 \dots n]$:

```
SUM-ARRAYS(A, B, C)
1 parallel for i = 1 to A.length
2     C[i] = A[i] + B[i]
```

- a.** Rewrite the parallel loop in SUM-ARRAYS using nested parallelism (**spawn** and **sync**) in the manner of MAT-VEC-MAIN-LOOP. Analyze the parallelism of your implementation.

```
MAT-VEC-MAIN-LOOP(A, B, C, l, r)
1 if l == r
2     C[l] = A[l] + B[l]
3 mid = (l + r) / 2
4 spawn MAT-VEC-MAIN-LOOP(A, B, C, l, mid)
5 MAT-VEC-MAIN-LOOP(A, B, C, mid + 1, r)
6 sync
```

```
SUM-ARRAYS(A, B, C)
1 len = A.length
2 MAT-VEC-MAIN-LOOP(A, B, C, 1, len)
```

Consider the following alternative implementation of the parallel loop, which contains a value grain-size to be specified:

- b.** Suppose that we set $grain\text{-}size = 1$. What is the parallelism of this implementation?

$$T_1 = \Theta(n), T_\infty = \Theta(n), T_1/T_\infty = \Theta(1)$$

- c.** Give a formula for the span of SUM-ARRAYS' in terms of n and $grain\text{-}size$. Derive the best value for grain-size to maximize parallelism.

$$T_\infty(n) = \Theta(\max(grain\text{-}size, n/grain\text{-}size))$$

27-2 Saving temporary space in matrix multiplication

The P-MATRIX-MULTIPLY-RECURSIVE procedure has the disadvantage that it must allocate a temporary matrix T of size $n \times n$, which can adversely affect the constants hidden by the Θ -notation. The P-MATRIX-MULTIPLY-RECURSIVE procedure does have high parallelism, however. For example, ignoring the constants in the Θ -notation, the parallelism for multiplying 1000×1000 matrices comes to approximately $1000^3 / 10^2 = 10^7$, since $\lg 1000 \approx 10$. Most parallel computers have far fewer than 10 million processors.

- a.** Describe a recursive multithreaded algorithm that eliminates the need for the temporary matrix T at the cost of increasing the span to $\Theta(n)$.

Initialize $C = 0$ in parallel in $\Theta(\lg n)$, add **sync** after the 4th **spawn**,
 $c_{11} = c_{11} + a_{11} \cdot b_{11}$, $T_\infty(n) = 2T_\infty(n/2) + \Theta(\lg n) = \Theta(n)$

- b.** Give and solve recurrences for the work and span of your implementation.

- Work: $T_1 = \Theta(n^3)$
- Span: $T_\infty = \Theta(n)$

- c.** Analyze the parallelism of your implementation. Ignoring the constants in the Θ -notation, estimate the parallelism on 1000×1000 matrices. Compare with the parallelism of P-MATRIX-MULTIPLY-RECURSIVE.

Parallelism: $T_1/T_\infty = \Theta(n^2) = 1000^2 = 10^6$

Most parallel computers still have far fewer than 1 million processors.

27-3 Multithreaded matrix algorithms

- a.** Parallelize the LU-DECOMPOSITION procedure on page 821 by giving pseudocode for a multithreaded version of this algorithm. Make your implementation as parallel as possible, and analyze its work, span, and parallelism.
- b.** Do the same for LUP-DECOMPOSITION on page 824.
- c.** Do the same for LUP-SOLVE on page 817.
- d.** Do the same for a multithreaded algorithm based on equation (28.13) for inverting a symmetric positive-definite matrix.

27-4 Multithreading reductions and prefix computations

A \otimes -**reduction** of an array $x[1 \dots n]$, where \otimes is an associative operator, is the value

$$y = x[1] \otimes x[2] \otimes \cdots \otimes x[n]$$

The following procedure computes the \otimes -reduction of a subarray $x[i \dots j]$ serially.

```
REDUCE(x, i, j)
1  y = x[i]
2  for k = i + 1 to j
3      y = y \otimes x[k]
4  return y
```

- a. Use nested parallelism to implement a multithreaded algorithm P-REDUCE, which performs the same function with $\Theta(n)$ work and $\Theta(\lg n)$ span. Analyze your algorithm.

```
REDUCE(x, i, j)
1  if i == j
2      return x[i]
3  else if i + 1 == j
4      return x[i] \otimes x[j]
5  mid = (i + j) / 2
6  spawn y1 = REDUCE(x, i, mid)
7  y2 = REDUCE(x, mid + 1, j)
8  sync
9  return y1 \otimes y2
```

A related problem is that of computing a \otimes -**prefix** computation, sometimes called a \otimes -**scan**, on an array $x[1 \dots n]$, where \otimes is once again an associative operator. The \otimes -scan produces the array $y[1 \dots n]$.

Unfortunately, multithreading SCAN is not straightforward. For example, changing the **for** loop to a **parallel for** loop would create races, since each iteration of the loop body depends on the previous iteration. The following procedure P-SCAN-1 performs the \otimes -prefix computation in parallel, albeit inefficiently.

- b. Analyze the work, span, and parallelism of P-SCAN-1.

- Work: $T_1 = \Theta(n^2)$
- Span: $T_\infty = \Theta(\lg n) + \Theta(\lg n) = \Theta(\lg n)$
- Parallelism: $T_1/T_\infty = \Theta(n^2/\lg n)$

By using nested parallelism, we can obtain a more efficient \otimes -prefix computation

c. Argue that P-SCAN-2 is correct, and analyze its work, span, and parallelism.

- Work: $T_1(n) = 2T_1(n/2) + \Theta(n) = \Theta(n \lg n)$
- Span: $T_\infty(n) = T_\infty(n/2) + \Theta(\lg n) = \Theta(\lg^2 n)$
- Parallelism: $T_1/T_\infty = \Theta(n/\lg n)$

d. Fill in the three missing expressions in line 8 of P-SCAN-UP and lines 5 and 6 of P-SCAN-DOWN. Argue that with expressions you supplied, P-SCAN-3 is correct.

- 8: $t[k] * \text{right}$
- 5: v
- 6: $t[k]$

e. Analyze the work, span, and parallelism of P-SCAN-3.

- Work: $T_1 = \Theta(n)$
- Span: $T_\infty = \Theta(\lg n)$
- Parallelism: $T_1/T_\infty = \Theta(n/\lg n)$

27-5 Multithreading a simple stencil calculation

Computational science is replete with algorithms that require the entries of an array to be filled in with values that depend on the values of certain already computed neighboring entries, along with other information that does not change over the course of the computation. The pattern of neighboring entries does not change during the computation and is called a **stencil**.

a. Give multithreaded pseudocode that performs this simple stencil calculation using a divide-and-conquer algorithm SIMPLE-STENCIL based on the decomposition (27.11) and the discussion above. (Don't worry about the details of the base case, which depends on the specific stencil.) Give and solve recurrences for the work and span of this algorithm in terms of n . What is the parallelism?

```

SIMPLE-STENCIL(A)
1 SIMPLE-STENCIL(A11)
2 spawn SIMPLE-STENCIL(A12)
3 SIMPLE-STENCIL(A21)
3 sync
5 SIMPLE-STENCIL(A22)
    
```

- **Work:** $T_1 = \Theta(n^2)$
- **Span:** $T_\infty(n) = 3T_\infty(n/2) + \Theta(1) = \Theta(n^{\lg 3}) \approx \Theta(n^{1.58})$
- **Parallelism:** $T_1/T_\infty = \Theta(n^{2/\lg 3}) \approx \Theta(n^{1.26})$

b. Modify your solution to part (a) to divide an $n \times n$ array into nine $n/3 \times n/3$ subarrays, again recursing with as much parallelism as possible. Analyze this algorithm. How much more or less parallelism does this algorithm have compared with the algorithm from part (a)?

```
11
spawn 12 21 sync
spawn 13 22 31 sync
spawn 23 32 sync
33
```

- **Work:** $T_1 = \Theta(n^2)$
- **Span:** $T_\infty(n) = 5T_\infty(n/3) + \Theta(1) = \Theta(n^{\log_3 5}) \approx \Theta(n^{1.46})$
- **Parallelism:** $T_1/T_\infty = \Theta(n^{2/\log_3 5}) \approx \Theta(n^{1.37})$

c. Generalize your solutions to parts (a) and (b) as follows. Choose an integer $b \geq 2$. Divide an $n \times n$ array into b^2 subarrays, each of size $n/b \times n/b$, recursing with as much parallelism as possible. In terms of n and b , what are the work, span, and parallelism of your algorithm? Argue that, using this approach, the parallelism must be $o(n)$ for any choice of $b \geq 2$. (Hint: For this last argument, show that the exponent of n in the parallelism is strictly less than 1 for any choice of $b \geq 2$.)

- **Work:** $T_1 = \Theta(n^2)$
- **Span:** $T_\infty(n) = (2b - 1)T_\infty(n/b) + \Theta(1) = \Theta(n^{\log_b(2b-1)})$
- **Parallelism:** $T_1/T_\infty = \Theta(n^{2/\log_b(2b-1)}) = \Theta(n^{\log_{2b-1} b^2})$

$b^2 \leq 2b - 1$, $(b - 1)^2 \leq 0$, since $b \geq 2$, the parallelism must be $o(n)$.

d. Give pseudocode for a multithreaded algorithm for this simple stencil calculation that achieves $\Theta(n \lg n)$ parallelism. Argue using notions of work and span that the problem, in fact, has $\Theta(n)$ inherent parallelism. As it turns out, the divide-and-conquer nature of our multithreaded pseudocode does not let us achieve this maximal parallelism.

27-6 Randomized multithreaded algorithms

Just as with ordinary serial algorithms, we sometimes want to implement randomized multithreaded algorithms. This problem explores how to adapt the various performance measures in order to handle the expected behavior of such algorithms. It also asks you to design and analyze a multithreaded algorithm for randomized quicksort.

- a.** Explain how to modify the work law (27.2), span law (27.3), and greedy scheduler bound (27.4) to work with expectations when T_P , T_1 , and T_∞ are all random variables.

$$\mathbb{E}[T_P] \geq \mathbb{E}[T_1]/P$$

$$\mathbb{E}[T_P] \geq \mathbb{E}[T_\infty]$$

$$\mathbb{E}[T_P] \leq \mathbb{E}[T_1]/P + \mathbb{E}[T_\infty]$$

- b.** Consider a randomized multithreaded algorithm for which 1% of the time we have $T_1 = 10^4$ and $T_{10,000} = 1$, but for 99% of the time we have $T_1 = T_{10,000} = 10^9$. Argue that the **speedup** of a randomized multithreaded algorithm should be defined as $\mathbb{E}[T_1]/\mathbb{E}[T_P]$, rather than $\mathbb{E}[T_1/T_P]$.

$$\mathbb{E}[T_1] \approx \mathbb{E}[T_{10,000}] \approx 9.9 \times 10^8, \mathbb{E}[T_1]/\mathbb{E}[T_P] = 1$$

$$\mathbb{E}[T_1/T_{10,000}] = 10^4 * 0.01 + 0.99 = 100.99$$

- c.** Argue that the **parallelism** of a randomized multithreaded algorithm should be defined as the ratio $\mathbb{E}[T_1]/\mathbb{E}[T_\infty]$.

Same as the above.

- d.** Multithread the RANDOMIZED-QUICKSORT algorithm on page 179 by using nested parallelism. (Do not parallelize RANDOMIZED-PARTITION.) Give the pseudocode for your P-RANDOMIZED-QUICKSORT algorithm.

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r
2      q = RANDOM-PARTITION(A, p, r)
3      spawn RANDOMIZED-QUICKSORT(A, p, q - 1)
4      RANDOMIZED-QUICKSORT(A, q + 1, r)
5      sync
```

- e.** Analyze your multithreaded algorithm for randomized quicksort. (Hint: Review the analysis of RANDOMIZED-SELECT on page 216.)

$$\mathbb{E}[T_1] = O(n \lg n)$$

$$\mathbb{E}[T_\infty] = O(\lg n)$$

$$\mathbb{E}[T_1]/\mathbb{E}[T_\infty] = O(n)$$

28 Matrix Operations

- 28.1 Solving systems of linear equations
- 28.2 Inverting matrices
- 28.3 Symmetric positive-definite matrices and least-squares approximation
- Problems

28.1 Solving systems of linear equations

28.2 Inverting matrices

28.3 Symmetric positive-definite matrices and least-squares approximation

Problems

29 Linear Programming

- 29.1 Standard and slack forms
- 29.2 Formulating problems as linear programs
- 29.3 The simplex algorithm
- 29.4 Duality
- 29.5 The initial basic feasible solution
- Problems

29.1 Standard and slack forms

29.2 Formulating problems as linear programs

29.3 The simplex algorithm

29.4 Duality

29.5 The initial basic feasible solution

Problems

30 Polynomials and the FFT

- 30.1 Representing polynomials
- 30.2 The DFT and FFT
- 30.3 Efficient FFT implementations
- Problems

30.1 Representing polynomials

30.2 The DFT and FFT

30.3 Efficient FFT implementations

Problems

31 Number-Theoretic Algorithms

- 31.1 Elementary number-theoretic notions
- 31.2 Greatest common divisor
- 31.3 Modular arithmetic
- 31.4 Solving modular linear equations
- 31.5 The Chinese remainder theorem
- 31.6 Powers of an element
- 31.7 The RSA public-key cryptosystem
- 31.8 Primality testing
- 31.9 Integer factorization
- Problems

31.1 Elementary number-theoretic notions

31.1-1

Prove that if $a > b > 0$ and $c = a + b$, then $c \bmod a = b$.

$$\begin{aligned} c \bmod a &= (a + b) \bmod a \\ &= (a \bmod a) + (b \bmod a) \\ &= 0 + b \\ &= b \end{aligned}$$

31.1-2

Prove that there are infinitely many primes.

$$\begin{aligned} &((p_1 p_2 \cdots p_k) + 1) \bmod p_i \\ &= (p_1 p_2 \cdots p_k) \bmod p_i + (1 \bmod p_i) \\ &= 0 + 1 \\ &= 1 \end{aligned}$$

31.1-3

Prove that if $a | b$ and $b | c$, then $a | c$.

- If $a | b$, then $b = a \cdot k_1$.
- If $b | c$, then $c = b \cdot k_2 = a \cdot (k_1 \cdot k_2) = a \cdot k_3$, then $a | c$.

31.1-4

Prove that if p is prime and $0 < k < p$, then $\gcd(k, p) = 1$.

- If $k \neq 1$, then $k \nmid p$.
- If $k = 1$, then the divisor is 1.

31.1-5

Prove Corollary 31.5.

For all positive integers n , a , and b , if $n \mid ab$ and $\gcd(a, n) = 1$, then $n \mid b$.

If $n \mid ab$, then $ab = kn$, then $b = nk/a$; since $\gcd(a, n) = 1$, then n/a could not be an integer; since b is an integer, then k/a must be an integer,
 $b = nk/a = n(k/a) = nk'$, then $n \mid b$.

31.1-6

Prove that if p is prime and $0 < k < p$, then $p \mid \binom{p}{k}$. Conclude that for all integers a and b and all primes p ,

$$(a + b)^p \equiv a^p + b^p \pmod{p}$$

$$\begin{aligned} (a + b)^p &\equiv a^p + \binom{p}{1}a^{p-1}b^1 + \cdots + \binom{p}{p-1}a^1b^{p-1} + b^p \pmod{p} \\ &\equiv a^p + 0 + \cdots + 0 + b^p \pmod{p} \\ &\equiv a^p + b^p \pmod{p} \end{aligned}$$

31.1-7

Prove that if a and b are any positive integers such that $a \mid b$, then

$$(x \bmod b) \bmod a = x \bmod a$$

for any x . Prove, under the same assumptions, that

$$x \equiv y \pmod{b} \text{ implies } x \equiv y \pmod{a}$$

for any integers x and y .

Suppose $x = kb + c$, then $(x \bmod b) \bmod a = c \bmod a$, and
 $x \bmod a = (kb + c) \bmod a = (kb \bmod a) + (c \bmod a) = c \bmod a$.

31.1-8

For any integer $k > 0$, an integer n is a **k th power** if there exists an integer a such that $a^k = n$. Furthermore, $n > 1$ is a **nontrivial power** if it is a k th power for some integer $k > 1$. Show how to determine whether a given β -bit integer n is a nontrivial power in time polynomial in β .

Iterate a from 2 to \sqrt{n} , and do binary searching.

31.1-9

Prove equations (31.6)-(31.10).

...

31.1-10

Show that the gcd operator is associative. That is, prove that for all integers a , b , and c ,

$$\gcd(a, \gcd(b, c)) = \gcd(\gcd(a, b), c)$$

...

31.1-11 *

Prove Theorem 31.8.

31.1-12

Give efficient algorithms for the operations of dividing a β -bit integer by a shorter integer and of taking the remainder of a β -bit integer when divided by a shorter integer. Your algorithms should run in time $\Theta(\beta^2)$.

Shift left until the two numbers have the same length, then repeatedly subtract with proper multiplier and shift right.

31.1-13

Give an efficient algorithm to convert a given β -bit (binary) integer to a decimal representation. Argue that if multiplication or division of integers whose length is at most β takes time $M(\beta)$, then we can convert binary to decimal in time $\Theta(M(\beta) \lg \beta)$.

```
def bin2dec(s):
    n = len(s)
    if n == 1:
        return ord(s) - ord('0')
    m = n // 2
    h = bin2dec(s[:m])
    l = bin2dec(s[m:])
    return (h << (n - m)) + l
```

31.2 Greatest common divisor

31.2-1

Prove that equations (31.11) and (31.12) imply equation (31.13).

31.2-2

Compute the values (d, x, y) that the call EXTENDED-EUCLID $(899, 493)$ returns.
 $(29, -6, 11)$

31.2-3

Prove that for all integers a , k , and n ,

$$\gcd(a, n) = \gcd(a + kn, n)$$

- $\gcd(a, n) \mid \gcd(a + kn, n)$

Let $d = \gcd(a, n)$, then $d \mid a$ and $d \mid n$. Since

$(a + kn) \bmod d = a \bmod d + k \cdot (n \bmod d) = 0$ and $d \mid n$, then
 $d \mid \gcd(a + kn, n)$, $\gcd(a, n) \mid \gcd(a + kn, n)$

- $\gcd(a + kn, n) \mid \gcd(a, n)$

Let $d = \gcd(a + kn, n)$, then $d \mid n$ and $d \mid (a + kn)$. Since

$(a + kn) \bmod d = a \bmod d + k \cdot (n \bmod d) = a \bmod d = 0$, then $d \mid a$.
Since $d \mid a$ and $d \mid n$, then $d \mid \gcd(a, n)$, $\gcd(a + kn, n) \mid \gcd(a, n)$.

Since $\gcd(a, n) \mid \gcd(a + kn, n)$ and $\gcd(a + kn, n) \mid \gcd(a, n)$, then
 $\gcd(a, n) = \gcd(a + kn, n)$

31.2-4

Rewrite EUCLID in an iterative form that uses only a constant amount of memory (that is, stores only a constant number of integer values).

```
def euclid(a, b):
    while b != 0:
        a, b = b, a % b
    return a
```

31.2-5

If $a > b \geq 0$, show that the call EUCLID (a, b) makes at most $1 + \log_\phi b$ recursive calls. Improve this bound to $1 + \log_\phi(b/\gcd(a, b))$.

$$b \geq F_{k+1} \approx \phi^{k+1}/\sqrt{5}$$

$$k + 1 < \log_\phi \sqrt{5} + \log_\phi b \approx 1.67 + \log_\phi b$$

$$k < 0.67 + \log_\phi b < 1 + \log_\phi b$$

Since $d \cdot a \bmod d \cdot b = d \cdot (a \bmod b)$, $\gcd(d \cdot a, d \cdot b)$ has the same number of recursive calls with $\gcd(a, b)$, therefore we could let $b' = b/\gcd(a, b)$, the inequality $k < 1 + \log_\phi(b') = 1 + \log_\phi(b/\gcd(a, b))$ will holds.

31.2-6

What does EXTENDED-EUCLID (F_{k+1}, F_k) return? Prove your answer correct.

- If k is odd, then $(1, -F_{k-2}, F_{k-1})$
- If k is even, then $(1, F_{k-2}, -F_{k-1})$

31.2-7

Define the \gcd function for more than two arguments by the recursive equation $\gcd(a_0, a_1, \dots, a_n) = \gcd(a_0, \gcd(a_1, a_2, \dots, a_n))$. Show that the \gcd function returns the same answer independent of the order in which its arguments are specified. Also show how to find integers x_0, x_1, \dots, x_n such that $\gcd(a_0, a_1, \dots, a_n) = a_0 x_0 + a_1 x_1 + \dots + a_n x_n$. Show that the number of divisions performed by your algorithm is $O(n + \lg(\max\{a_0, a_1, \dots, a_n\}))$.

Suppose $\gcd(a_0, \gcd(a_1, a_2, \dots, a_n)) = a_0 \cdot x + \gcd(a_1, a_2, \dots, a_n) \cdot y$ and $\gcd(a_1, \gcd(a_2, a_3, \dots, a_n)) = a_1 \cdot x' + \gcd(a_2, a_3, \dots, a_n) \cdot y'$, then the coefficient of a_1 is $y \cdot x'$.

```
def extended_euclid(a, b):
    if b == 0:
        return (a, 1, 0)
    d, x, y = extended_euclid(b, a % b)
    return (d, y, x - (a // b) * y)

def extended_eculid_multi(a):
    if len(a) == 1:
        return (a[0], [1])
    g = a[-1]
    xs = [1] * len(a)
    ys = [0] * len(a)
    for i in xrange(len(a) - 2, -1, -1):
        g, xs[i], ys[i + 1] = extended_euclid(a[i], g)
    m = 1
    for i in xrange(1, len(a)):
        m *= ys[i]
        xs[i] *= m
    return (g, xs)
```

31.2-8

Define $\text{lcm}(a_1, a_2, \dots, a_n)$ to be the *least common multiple* of the n integers a_1, a_2, \dots, a_n , that is, the smallest nonnegative integer that is a multiple of each a_i . Show how to compute $\text{lcm}(a_1, a_2, \dots, a_n)$ efficiently using the (two-argument) \gcd operation as a subroutine.

```

def gcd(a, b):
    if b == 0:
        return a
    return gcd(b, a % b)

def lcm(a, b):
    return a / gcd(a, b) * b

def lcm_multi(lst):
    l = lst[0]
    for i in xrange(1, len(lst)):
        l = lcm(l, lst[i])
    return l

```

31.2-9

Prove that n_1, n_2, n_3 , and n_4 are pairwise relatively prime if and only if

$$\gcd(n_1n_2, n_3n_4) = \gcd(n_1n_3, n_2n_4) = 1$$

More generally, show that n_1, n_2, \dots, n_k are pairwise relatively prime if and only if a set of $\lceil \lg k \rceil$ pairs of numbers derived from the n_i are relatively prime.

Suppose $n_1n_2x + n_3n_4y = 1$, then $n_1(n_2x) + n_3(n_4y) = 1$, thus n_1 and n_3 are relatively prime, n_1 and n_4 , n_2 and n_3 , n_2 and n_4 are all relatively prime. And since $\gcd(n_1n_3, n_2n_4) = 1$, all the pairs are relatively prime.

General: in the first round, divide the elements into two sets with equal number of elements, calculate the products of the two set separately, if the two products are relatively prime, then the element in one set is pairwise relatively prime with the element in the other set. In the next iterations, for each set, we divide the elements into two subsets, suppose we have

subsets $\{(s_1, s_2), (s_3, s_4), \dots\}$, then we calculate the products of $\{s_1, s_3, \dots\}$ and $\{s_2, s_4, \dots\}$, if the two products are relatively prime, then all the pairs of subset are pairwise relatively prime similar to the first round. In each iteration, the number of elements in a subset is half of the original set, thus there are $\lceil \lg k \rceil$ pairs of products.

To choose the subsets efficiently, in the k th iteration, we could divide the numbers based on the value of the index's k th bit.

31.3 Modular arithmetic

31.3-1

Draw the group operation tables for the groups $(\mathbb{Z}_4, +_4)$ and $(\mathbb{Z}_5^*, \cdot_5)$. Show that these groups are isomorphic by exhibiting a one-to-one correspondence α between their elements such that $a + b \equiv c \pmod{4}$ if and only if $\alpha(a) \cdot \alpha(b) \equiv \alpha(c) \pmod{5}$.

- $(\mathbb{Z}_4, +_4), \{0, 1, 2, 3\}$
- $(\mathbb{Z}_5^*, \cdot_5), \{1, 2, 3, 4\}$

$$\alpha(x) = 2^{x-1}$$

31.3-2

List all subgroups of \mathbb{Z}_9 and of \mathbb{Z}_{13}^* .

- \mathbb{Z}_9
 - ,
 - ,
 - ,
 - ,
- \mathbb{Z}_{13}^*
 - ,
 - ,
 - ,

31.3-3

Prove Theorem 31.14.

A nonempty closed subset of a finite group is a subgroup.

- Closure: the subset is closed.
- Identity: suppose $a \in S'$, then $a^{(k)} \in S'$. Since the subset is finite, there must be a period such that $a^{(m)} = a^{(n)}$, hence $a^{(m)}a^{(-n)} = a^{(m-n)} = 1$, therefore the subset must contain the identity.

- Associativity: inherit from the origin group.
- Inverses: suppose $a^{(k)} = 1$, the inverse of element a is $a^{(k-1)}$ since $aa^{(k-1)} = a^{(k)} = 1$.

31.3-4

Show that if p is prime and e is a positive integer, then

$$\phi(p^e) = p^{e-1}(p - 1)$$

$$\phi(p^e) = p^e \cdot \left(1 - \frac{1}{p}\right) = p^{e-1}(p - 1)$$

31.3-5

Show that for any integer $n > 1$ and for any $a \in \mathbb{Z}_n^*$, the function $f_a : \mathbb{Z}_n^* \rightarrow \mathbb{Z}_n^*$ defined by $f_a(x) = ax \bmod n$ is a permutation of \mathbb{Z}_n^* .

To prove it is a permutation, we need to prove that

- for each element $x \in \mathbb{Z}_n^*$, $f_a(x) \in \mathbb{Z}_n^*$,
- the numbers generated by f_a are distinct.

Since $a \in \mathbb{Z}_n^*$ and $x \in \mathbb{Z}_n^*$, then $f_a(x) = ax \bmod n \in \mathbb{Z}_n^*$ by the closure property.

Suppose there are two distinct numbers $x \in \mathbb{Z}_n^*$ and $y \in \mathbb{Z}_n^*$ that $f_a(x) = f_a(y)$,

$$\begin{aligned} f_a(x) &= f_a(y) \\ ax \bmod n &= ay \bmod n \\ (a \bmod n)(x \bmod n) &= (a \bmod n)(y \bmod n) \\ (x \bmod n) &= (y \bmod n) \\ x &\equiv y \quad (\bmod n) \end{aligned}$$

which contradicts the assumption, therefore the numbers generated by f_a are distinct.

31.4 Solving modular linear equations

31.4-1

Find all solutions to the equation $35x \equiv 10 \pmod{50}$

$$\{6, 16, 26, 36, 46\}$$

31.4-2

Prove that the equation $ax \equiv ay \pmod{n}$ implies $x \equiv y \pmod{n}$ whenever $\gcd(a, n) = 1$. Show that the condition $\gcd(a, n) = 1$ is necessary by supplying a counterexample with $\gcd(a, n) > 1$.

$$d = \gcd(ax, n) = \gcd(x, n)$$

Since $ax \cdot x' + n \cdot y' = d$, then $x \cdot (ax') + n \cdot y' = d$.

$$x_0 = x'(ay/d)$$

$$x'_0 = (ax')(y/d) = x'(ay/d) = x_0$$

31.4-3

Consider the following change to line 3 of the procedure MODULAR-LINEAR-EQUATION-SOLVER:

$$3 \quad x_0 = x'(b/d) \bmod (n/d)$$

Assume that $x_0 \geq n/d$, then the largest solution is $x_0 + (d - 1) * (n/d) \geq d * n/d \geq n$, which is impossible, therefore $x_0 < n/d$.

31.4-4 *

Let p be prime and $f(x) \equiv f_0 + f_1x + \cdots + f_tx^t \pmod{p}$ be a polynomial of degree t , with coefficients f_i drawn from \mathbb{Z}_p . We say that $a \in \mathbb{Z}_p$ is a **zero** of f if $f(a) \equiv 0 \pmod{p}$. Prove that if a is a zero of f , then $f(x) \equiv (x - a)g(x) \pmod{p}$ for some polynomial $g(x)$ of degree $t - 1$. Prove by induction on t that if p is prime, then a polynomial $f(x)$ of degree t can have at most t distinct zeros modulo p .

31.5 The Chinese remainder theorem

31.5-1

Find all solutions to the equations $x \equiv 4 \pmod{5}$ and $x \equiv 5 \pmod{11}$.

$$m_1 = 11, m_2 = 5$$

$$m_1^{-1} = 1, m_2^{-1} = 9$$

$$c_1 = 11, c_2 = 45$$

$$a = (c_1 \cdot a_1 + c_2 \cdot a_2) \pmod{(n_1 \cdot n_2)} = (11 * 4 + 45 * 5) \pmod{55} = 49$$

31.5-2

Find all integers x that leave remainders 1, 2, 3 when divided by 9, 8, 7 respectively.

$$10 + 504i, i \in \mathbb{Z}$$

31.5-3

Argue that, under the definitions of Theorem 31.27, if $\gcd(a, n) = 1$, then

$$(a^{-1} \pmod{n}) \leftrightarrow ((a_1^{-1} \pmod{n_1}), (a_2^{-1} \pmod{n_2}), \dots, (a_k^{-1} \pmod{n_k}))$$

$$\gcd(a, n) = 1 \rightarrow \gcd(a, n_i) = 1$$

31.5-4

Under the definitions of Theorem 31.27, prove that for any polynomial f , the number of roots of the equation $f(x) \equiv 0 \pmod{n}$ equals the product of the number of roots of each of the equations

$$f(x) \equiv 0 \pmod{n_1}, f(x) \equiv 0 \pmod{n_2}, \dots, f(x) \equiv 0 \pmod{n_k}$$

Based on 31.28 ~ 31.30.

31.6 Powers of an element

31.6-1

Draw a table showing the order of every element in \mathbb{Z}_{11}^* . Pick the smallest primitive root g and compute a table giving $\text{ind}_{11,g}(x)$ for all $x \in \mathbb{Z}_{11}^*$.

$$g = 2, \{1, 2, 4, 8, 5, 10, 9, 7, 3, 6\}$$

31.6-2

Give a modular exponentiation algorithm that examines the bits of b from right to left instead of left to right.

```
def modular_exponentiation(a, b, n):
    i, d = 0, 1
    while (1 << i) <= b:
        if (b & (1 << i)) > 0:
            d = (d * a) % n
        a = (a * a) % n
        i += 1
    return d
```

31.6-3

Assuming that you know $\phi(n)$, explain how to compute $a^{-1} \bmod n$ for any $a \in \mathbb{Z}_n^*$ using the procedure MODULAR-EXPONENTIATION.

$$\begin{aligned} a^{\phi(n)} &\equiv 1 \pmod{n} \\ a \cdot a^{\phi(n)-1} &\equiv 1 \pmod{n} \\ a^{-1} &\equiv a^{\phi(n)-1} \pmod{n} \end{aligned}$$

31.7 The RSA public-key cryptosystem

31.7-1

Consider an RSA key set with $p = 11$, $q = 29$, $n = 319$, and $e = 3$. What value of d should be used in the secret key? What is the encryption of the message $M = 100$?

$$\phi(n) = (p - 1) \cdot (q - 1) = 280$$

$$d = e^{-1} \bmod \phi(n) = 187$$

$$P(M) = M^e \bmod n = 254$$

$$S(C) = C^d \bmod n = 254^{187} \bmod n = 100$$

31.7-2

Prove that if Alice's public exponent e is 3 and an adversary obtains Alice's secret exponent d , where $0 < d < \phi(n)$, then the adversary can factor Alice's modulus n in time polynomial in the number of bits in n . (Although you are not asked to prove it, you may be interested to know that this result remains true even if the condition $e = 3$ is removed. See Miller [255].)

$$ed \equiv 1 \bmod \phi(n)$$

$$ed - 1 = 3d - 1 = k\phi(n)$$

$$\text{If } p, q < n/4, \text{ then } \phi(n) = n - (p + q) + 1 > n - n/2 + 1 = n/2 + 1 > n/2$$

$$kn/2 < 3d - 1 < 3d < 3n, \text{ then } k < 6, \text{ then we can solve}$$

$$3d - 1 = n - p - n/p + 1$$

31.7-3 *

Prove that RSA is multiplicative in the sense that

$$P_A(M_1)P_A(M_2) \equiv P_A(M_1, M_2) \pmod{n}$$

Use this fact to prove that if an adversary had a procedure that could efficiently decrypt 1 percent of messages from \mathbb{Z}_n encrypted with P_A , then he could employ a probabilistic algorithm to decrypt every message encrypted with P_A with high probability.

Multiplicative: e is relatively prime to n .

Decrypt: In each iteration randomly choose a prime number m that m is relatively prime to n , if we can decrypt $m \cdot M$, then we can return $m^{-1}M$ since $m^{-1} = m^{n-2}$.

31.8 Primality testing

31.8-1

Prove that if an odd integer $n > 1$ is not a prime or a prime power, then there exists a nontrivial square root of 1 modulo n .

31.8-2 *

31.8-3

Prove that if x is a nontrivial square root of 1, modulo n , then $\gcd(x - 1, n)$ and $\gcd(x + 1, n)$ are both nontrivial divisors of n .

$$\begin{aligned}x^2 &\equiv 1 \pmod{n} \\x^2 - 1 &\equiv 0 \pmod{n} \\(x + 1)(x - 1) &\equiv 0 \pmod{n}\end{aligned}$$

$n \mid (x + 1)(x - 1)$, suppose $\gcd(x - 1, n) = 1$, then $n \mid (x + 1)$, then $x \equiv -1 \pmod{n}$ which is trivial, it contradicts the fact that x is nontrivial, therefore $\gcd(x - 1, n) \neq 1$, $\gcd(x + 1, n) \neq 1$.

31.9 Integer factorization

31.9-1

Referring to the execution history shown in Figure 31.7(a), when does POLLARDRHO print the factor 73 of 1387?

$$x = 84, y = 814$$

31.9-2

Suppose that we are given a function $f : \mathbb{Z}_n \rightarrow \mathbb{Z}_n$ and an initial value $x_0 \in \mathbb{Z}_n$. Define $x_i = f(x_{i-1})$ for $i = 1, 2, \dots$. Let t and $u > 0$ be the smallest values such that $x_{t+i} = x_{t+u+i}$ for $i = 0, 1, \dots$. In the terminology of Pollard's rho algorithm, t is the length of the tail and u is the length of the cycle of the rho. Give an efficient algorithm to determine t and u exactly, and analyze its running time.

31.9-3

How many steps would you expect POLLARD-RHO to require to discover a factor of the form p^e , where p is prime and $e > 1$?

$$\Theta(\sqrt{p})$$

31.9-4 *

One disadvantage of POLLARD-RHO as written is that it requires one gcd computation for each step of the recurrence. Instead, we could batch the gcd computations by accumulating the product of several x_i values in a row and then using this product instead of x_i in the gcd computation. Describe carefully how you would implement this idea, why it works, and what batch size you would pick as the most effective when working on a β -bit number n .

Problems

31-1 Binary gcd algorithm

Most computers can perform the operations of subtraction, testing the parity (odd or even) of a binary integer, and halving more quickly than computing remainders. This problem investigates the *binary gcd algorithm*, which avoids the remainder computations used in Euclid's algorithm.

- Prove that if a and b are both even, then $\gcd(a, b) = 2 \cdot \gcd(a/2, b/2)$.
- Prove that if a is odd and b is even, then $\gcd(a, b) = \gcd(a, b/2)$.
- Prove that if a and b are both odd, then $\gcd(a, b) = \gcd((a - b)/2, b)$.
- Design an efficient binary gcd algorithm for input integers a and b , where $a \geq b$, that runs in $O(\lg a)$ time. Assume that each subtraction, parity test, and halving takes unit time.

```
def binary_gcd(a, b):
    if a < b:
        return binary_gcd(b, a)
    if b == 0:
        return a
    if (a & 1 == 1) and (b & 1 == 1):
        return binary_gcd((a - b) >> 1, b)
    if (a & 1 == 0) and (b & 1 == 0):
        return binary_gcd(a >> 1, b >> 1) << 1
    if a & 1 == 1:
        return binary_gcd(a, b >> 1)
    return binary_gcd(a >> 1, b)
```

31-2 Analysis of bit operations in Euclid's algorithm

- Consider the ordinary "paper and pencil" algorithm for long division: dividing a by b , which yields a quotient q and remainder r . Show that this method requires $O((1 + \lg q) \lg b)$ bit operations.

Number of comparisons and subtractions: $\lceil \lg a \rceil - \lceil \lg b \rceil + 1 = \lceil \lg q \rceil$

Length of subtraction: $\lceil \lg b \rceil$.

Total: $O((1 + \lg q) \lg b)$

b. Define $\mu(a, b) = (1 + \lg a)(1 + \lg b)$. Show that the number of bit operations performed by EUCLID in reducing the problem of computing $\gcd(a, b)$ to that of computing $\gcd(b, a \bmod b)$ is at most $c(\mu(a, b) - \mu(b, a \bmod b))$ for some sufficiently large constant $c > 0$.

$$\begin{aligned}
 & \mu(a, b) - \mu(b, a \bmod b) \\
 = & \mu(a, b) - \mu(b, r) \\
 = & (1 + \lg a)(1 + \lg b) - (1 + \lg b)(1 + \lg r) \\
 = & (1 + \lg b)(\lg a - \lg r) \quad (\lg r \leq \lg b) \\
 \geq & (1 + \lg b)(\lg a - \lg b) \\
 = & (1 + \lg b)(\lg q + 1) \\
 \geq & (1 + \lg q) \lg b
 \end{aligned}$$

c. Show that EUCLID (a, b) requires $O(\mu(a, b))$ bit operations in general and $O(\beta^2)$ bit operations when applied to two β -bit inputs.

$$\mu(a, b) = (1 + \lg a)(1 + \lg b) \approx \beta^2$$

31-3 Three algorithms for Fibonacci numbers

This problem compares the efficiency of three methods for computing the n th Fibonacci number F_n , given n . Assume that the cost of adding, subtracting, or multiplying two numbers is $O(1)$, independent of the size of the numbers.

a. Show that the running time of the straightforward recursive method for computing F_n based on recurrence (3.22) is exponential in n . (See, for example, the FIB procedure on page 775.)

$$T(n) = T(n-1) + T(n-2) + \Theta(1) = \Theta(2^n)$$

b. Show how to compute F_n in $O(n)$ time using memoization.

```

def fib(n):
    fibs = [0, 1] + [-1] * (n - 1)

    def fib_sub(n):
        if fibs[n] == -1:
            fibs[n] = fib_sub(n - 1) + fib_sub(n - 2)
        return fibs[n]
    return fib_sub(n)

```

- c. Show how to compute F_n in $O(\lg n)$ time using only integer addition and multiplication. (Hint: Consider the matrix

$$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

and its powers.)

```

class Matrix:
    def __init__(self, data):
        self.data = data

    def __mul__(self, x):
        a = self.data
        b = x.data
        c = [[0, 0], [0, 0]]
        for i in xrange(2):
            for j in xrange(2):
                for k in xrange(2):
                    c[i][j] += a[i][k] * b[k][j]
        return Matrix(c)

def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    m = Matrix([[1, 1], [1, 0]])
    r = Matrix([[1, 0], [0, 1]])
    i = 0
    n -= 1
    while (1 << i) <= n:
        if (n & (1 << i)) > 0:
            r *= m
        m *= m
        i += 1
    return r.data[0][0]

```

d. Assume now that adding two β -bit numbers takes $\Theta(\beta)$ time and that multiplying two β -bit numbers takes $\Theta(\beta^2)$ time. What is the running time of these three methods under this more reasonable cost measure for the elementary arithmetic operations?

1. $\Theta(2^{2^\beta})$
2. $\Theta(2^\beta)$
3. $T(n) = T(n/2) + \Theta(\beta^2) = \Theta(\beta^3)$

31-4 Quadratic residues

Let p be an odd prime. A number $a \in \mathbb{Z}_p^*$ is a **quadratic residue** if the equation $x^2 = a \pmod{p}$ has a solution for the unknown x .

- a.** Show that there are exactly $(p - 1)/2$ quadratic residues, modulo p .
- b.** If p is prime, we define the **Legendre symbol** $\left(\frac{a}{p}\right)$, for $a \in \mathbb{Z}_p^*$, to be 1 if a is a quadratic residue modulo p and -1 otherwise. Prove that if $a \in \mathbb{Z}_p^*$, then

$$\left(\frac{a}{p}\right) \equiv a^{(p-1)/2} \pmod{p}$$

Give an efficient algorithm that determines whether a given number a is a quadratic residue modulo p . Analyze the efficiency of your algorithm.

- c.** Prove that if p is a prime of the form $4k + 3$ and a is a quadratic residue in \mathbb{Z}_p^* , then $a^{k+1} \pmod{p}$ is a square root of a , modulo p . How much time is required to find the square root of a quadratic residue a modulo p ?
- d.** Describe an efficient randomized algorithm for finding a nonquadratic residue, modulo an arbitrary prime p , that is, a member of \mathbb{Z}_p^* that is not a quadratic residue. How many arithmetic operations does your algorithm require on average?

32 String Matching

- 32.1 The naive string-matching algorithm
- 32.2 The Rabin-Karp algorithm
- 32.3 String matching with finite automata
- 32.4 The Knuth-Morris-Pratt algorithm
- Problems

32.1 The naive string-matching algorithm

32.1-1

Show the comparisons the naive string matcher makes for the pattern $P = 0001$ in the text $T = 000010001010001$.

...

32.1-2

Suppose that all characters in the pattern P are different. Show how to accelerate NAIVE-STRING-MATCHER to run in time $O(n)$ on an n -character text T .

Suppose $T[i] \neq P[j]$, then for $k \in [1, j)$, $T[i - k] = P[j - k] \neq P[0]$, the $[i - k, i)$ are all invalid shifts which could be skipped, therefore we can compare $T[i]$ with $P[0]$ in the next iteration.

32.1-3

Suppose that pattern P and text T are randomly chosen strings of length m and n , respectively, from the d -ary alphabet $\Sigma_d = \{0, 1, \dots, d-1\}$, where $d \geq 2$. Show that the expected number of character-to-character comparisons made by the implicit loop in line 4 of the naive algorithm is

$$(n - m + 1) \frac{1 - d^{-m}}{1 - d^{-1}} \leq 2(n - m + 1)$$

over all executions of this loop. (Assume that the naive algorithm stops comparing characters for a given shift once it finds a mismatch or matches the entire pattern.) Thus, for randomly chosen strings, the naive algorithm is quite efficient.

Suppose for each shift, the number of compared characters is L , then:

$$\begin{aligned} E[L] &= 1 \cdot \frac{d-1}{d} + 2 \cdot \left(\frac{1}{d}\right)^1 \frac{d-1}{d} + \dots + m \cdot \left(\frac{1}{d}\right)^{m-1} \frac{d-1}{d} + m \cdot \left(\frac{1}{d}\right)^m \\ &= \left(1 + 2 \cdot \left(\frac{1}{d}\right)^1 + \dots + m \cdot \left(\frac{1}{d}\right)^m\right) \frac{d-1}{d} + m \cdot \left(\frac{1}{d}\right)^m \end{aligned}$$

$$\begin{aligned}
 S &= 1 + 2 \cdot \left(\frac{1}{d}\right)^1 + \cdots + m \cdot \left(\frac{1}{d}\right)^{m-1} \\
 \frac{1}{d}S &= 1 \cdot \left(\frac{1}{d}\right)^1 + \cdots + (m-1) \cdot \left(\frac{1}{d}\right)^{m-1} + m \cdot \left(\frac{1}{d}\right)^m \\
 \frac{d-1}{d}S &= 1 + \left(\frac{1}{d}\right)^1 + \cdots + \left(\frac{1}{d}\right)^{m-1} - m \cdot \left(\frac{1}{d}\right)^m \\
 \frac{d-1}{d}S &= \frac{1 - d^{-m}}{1 - d^{-1}} - m \cdot \left(\frac{1}{d}\right)^m \\
 \mathbb{E}[L] &= \left(1 + 2 \cdot \left(\frac{1}{d}\right)^1 + \cdots + m \cdot \left(\frac{1}{d}\right)^m\right) \frac{d-1}{d} + m \cdot \left(\frac{1}{d}\right)^m \\
 &= \frac{1 - d^{-m}}{1 - d^{-1}} - m \cdot \left(\frac{1}{d}\right)^m + m \cdot \left(\frac{1}{d}\right)^m \\
 &= \frac{1 - d^{-m}}{1 - d^{-1}}
 \end{aligned}$$

There are $n - m + 1$ shifts, therefore the expected number of comparisons is:

$$\begin{aligned}
 (n - m + 1) \cdot \mathbb{E}[L] &= (n - m + 1) \frac{1 - d^{-m}}{1 - d^{-1}} \\
 &\quad \text{And since } d \geq 2, 1 - d^{-1} \geq 0.5, \text{ and since } 1 - d^{-m} < 1, \frac{1 - d^{-m}}{1 - d^{-1}} \leq 2, \\
 &\quad \text{therefore } (n - m + 1) \frac{1 - d^{-m}}{1 - d^{-1}} \leq 2(n - m + 1).
 \end{aligned}$$

32.1-4

Suppose we allow the pattern P to contain occurrences of a **gap character** \diamond that can match an *arbitrary* string of characters (even one of zero length). For example, the pattern $ab \diamond ba \diamond c$ occurs in the text $cabccbacbacab$ as

$c \underbrace{ab}_{ab} \underbrace{cc}_{\diamond} \underbrace{ba}_{ba} \underbrace{cba}_{\diamond} \underbrace{c}_{c} \underbrace{ab}_{ab}$

and as

$c \underbrace{ab}_{ab} \underbrace{ccbac}_{\diamond} \underbrace{ba}_{ba} \underbrace{\diamond}_{c} \underbrace{c}_{ab} \underbrace{ab}_{ab}$

Note that the gap character may occur an arbitrary number of times in the pattern but not at all in the text. Give a polynomial-time algorithm to determine whether such a pattern P occurs in a given text T , and analyze the running time of your algorithm.

Dynamic programming, $\Theta(n^2m)$.

32.2 The Rabin-Karp algorithm

32.2-1

Working modulo $q = 11$, how many spurious hits does the Rabin-Karp matcher encounter in the text $T = 3141592653589793$ when looking for the pattern $P = 26$?

$$|\{15, 59, 92, 26\}| = 4$$

32.2-2

How would you extend the Rabin-Karp method to the problem of searching a text string for an occurrence of any one of a given set of k patterns? Start by assuming that all k patterns have the same length. Then generalize your solution to allow the patterns to have different lengths.

Truncation.

32.2-3

Show how to extend the Rabin-Karp method to handle the problem of looking for a given $m \times m$ pattern in an $n \times n$ array of characters. (The pattern may be shifted vertically and horizontally, but it may not be rotated.)

Calculate the hashes in each column just like the Rabin-Karp in one-dimension, then treat the hashes in each row as the characters and hashing again.

32.2-4

Alice has a copy of a long n -bit file $A = \langle a_{n-1}, a_{n-2}, \dots, a_0 \rangle$, and Bob similarly has an n -bit file $B = \langle b_{n-1}, b_{n-2}, \dots, b_0 \rangle$. Alice and Bob wish to know if their files are identical. To avoid transmitting all of A or B , they use the following fast probabilistic check. Together, they select a prime $q > 1000n$ and randomly select an integer x from $\{0, 1, \dots, q - 1\}$. Then, Alice evaluates

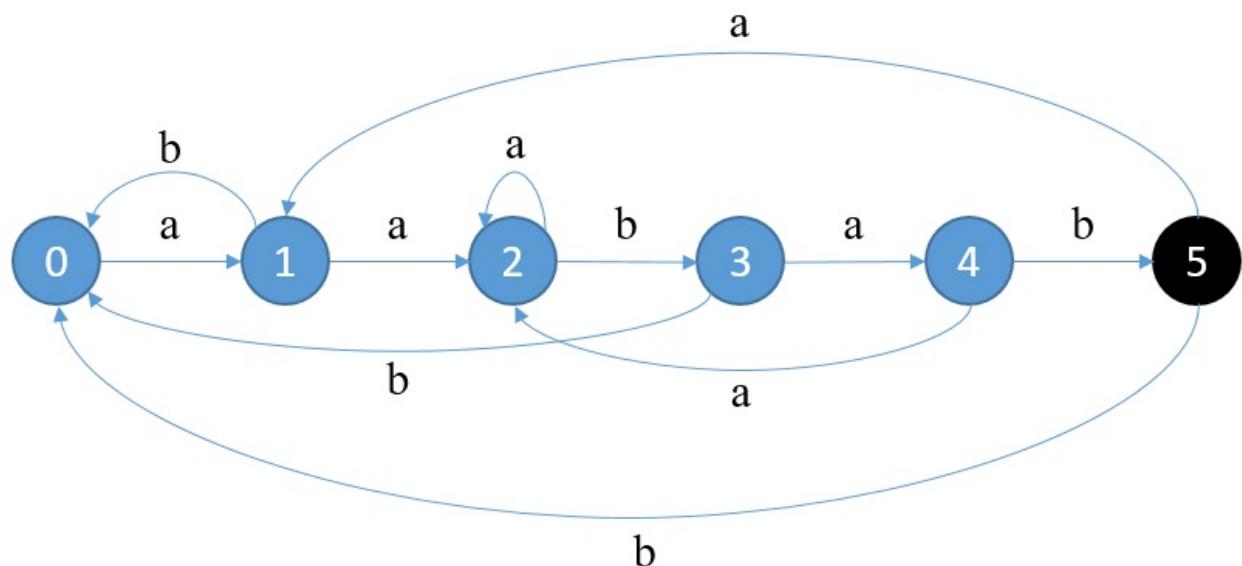
$$A(x) = \left(\sum_{i=0}^{n-1} a_i x^i \right) \bmod q$$

and Bob similarly evaluates $B(x)$. Prove that if $A \neq B$, there is at most one chance in 1000 that $A(x) = B(x)$, whereas if the two files are the same, $A(x)$ is necessarily the same as $B(x)$. (Hint: See Exercise 31.4-4.)

32.3 String matching with finite automata

32.3-1

Construct the string-matching automaton for the pattern $P = aabab$ and illustrate its operation on the text string $T = aaababaabaababaab$.



$0 \rightarrow 1 \rightarrow 2 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 1 \rightarrow 2 \rightarrow 3$.

32.3-2

Draw a state-transition diagram for a string-matching automaton for the pattern $ababbabbababbababbabb$ over the alphabet $\sigma = \{a, b\}$.

State	a	b
0	1	0
1	1	2
2	3	0
3	1	4
4	3	5
5	6	0
6	1	7
7	3	8
8	9	0
9	1	10
10	11	0
11	1	12
12	3	13
13	14	0
14	1	15
15	16	8
16	1	17
17	3	18
18	19	0
19	1	20
20	3	21
21	9	0

32.3-3

We call a pattern P *nonoverlappable* if $P_k \sqsupseteq P_q$ implies $k = 0$ or $k = q$. Describe the state-transition diagram of the string-matching automaton for a nonoverlappable pattern.

$$\delta(q, a) \in \{q + 1, 0\}$$

32.3-4 ★

Given two patterns P and P' , describe how to construct a finite automaton that determines all occurrences of either pattern. Try to minimize the number of states in your automaton.

Combine the common prefix and suffix.

32.3-5

Given a pattern P containing gap characters (see Exercise 32.1-4), show how to build a finite automaton that can find an occurrence of P in a text T in $O(n)$ matching time, where $n = |T|$.

Split the string with the gap characters, build finite automatons for each substring. When a substring is matched, moved to the next finite automaton.

32.4 The Knuth-Morris-Pratt algorithm

32.4-1

Compute the prefix function π for the pattern ababbabbabbabbabbabb.

$$\pi = \{0, 0, 1, 2, 0, 1, 2, 0, 1, 2, 0, 1, 2, 3, 4, 5, 6, 7, 8\}$$

32.4-2

Give an upper bound on the size of $\pi^*[q]$ as a function of q . Give an example to show that your bound is tight.

$$|\pi^*[q]| < q$$

32.4-3

Explain how to determine the occurrences of pattern P in the text T by examining the π function for the string PT (the string of length $m + n$ that is the concatenation of P and T).

$$\{q \mid \pi[q] = m \text{ and } q \geq 2m\}$$

32.4-4

Use an aggregate analysis to show that the running time of KMP-MATCHER is $\Theta(n)$.

The number of $q = q + 1$ is at most n .

32.4-5

Use a potential function to show that the running time of KMP-MATCHER is $\Theta(n)$.

$$\Phi = p$$

32.4-6

Show how to improve KMP-MATCHER by replacing the occurrence of π in line 7 (but not line 12) by π' , where π' is defined recursively for $q = 1, 2, \dots, m - 1$ by the equation

$$\pi'[q] = \begin{cases} 0 & \text{if } \pi[q] = 0, \\ \pi'[\pi[q]] & \text{if } \pi[q] \neq 0 \text{ and } P[\pi[q] + 1] = P[q + 1] \\ \pi[q] & \text{if } \pi[q] \neq 0 \text{ and } P[\pi[q] + 1] \neq P[q + 1] \end{cases}$$

Explain why the modified algorithm is correct, and explain in what sense this change constitutes an improvement.

If $P[q + 1] \neq T[i]$, then if $P[\pi[q] + q] = P[q + 1] \neq T[i]$, there is no need to compare $P[\pi[q] + q]$ with $T[i]$.

32.4-7

Give a linear-time algorithm to determine whether a text T is a cyclic rotation of another string T' . For example, `arc` and `car` are cyclic rotations of each other.

Find T' in TT .

32.4-8 *

Give an $O(m|\Sigma|)$ -time algorithm for computing the transition function δ for the string-matching automaton corresponding to a given pattern P . (Hint: Prove that $\delta(q, a) = \delta(\pi[q], a)$ if $q = m$ or $P[q + 1] \neq a$.)

Compute the prefix function m times.

Problems

32-1 String matching based on repetition factors

Let y^i denote the concatenation of string y with itself i times. For example,

$(ab)^3 = ababab$. We say that a string $x \in \Sigma^*$ has **repetition factor r** if $x = y^r$ for some string $y \in \Sigma^*$ and some $r > 0$. Let ρ denote the largest r such that x has repetition factor r .

- a. Give an efficient algorithm that takes as input a pattern $P[1 \dots m]$ and computes the value $\rho(P_i)$ for $i = 1, 2, \dots, m$. What is the running time of your algorithm?

Compute π , let $l = m - \pi[m]$, if $m \bmod l = 0$ and for all $p = m - i \cdot l > 0$, $p - \pi[p] = l$, then $\rho(P_i) = m/l$, otherwise $\rho(P_i) = 1$. The running time is $\Theta(n)$.

- b. For any pattern $P[1 \dots m]$, let $\rho^*(P)$ be defined as $\max_{1 \leq i \leq m} \rho(P_i)$. Prove that if the pattern P is chosen randomly from the set of all binary strings of length m , then the expected value of $\rho^*(P)$ is $O(1)$.

$$P(\rho^*(P) \geq 2) = \frac{1}{2} + \frac{1}{8} + \frac{1}{32} + \dots \approx \frac{2}{3}$$

$$P(\rho^*(P) \geq 3) = \frac{1}{4} + \frac{1}{32} + \frac{1}{256} + \dots \approx \frac{2}{7}$$

$$P(\rho^*(P) \geq 4) = \frac{1}{8} + \frac{1}{128} + \frac{1}{2048} + \dots \approx \frac{2}{15}$$

$$P(\rho^*(P) = 1) = \frac{1}{3}$$

$$P(\rho^*(P) = 2) = \frac{8}{21}$$

$$P(\rho^*(P) = 3) = \frac{16}{105}$$

$$E[\rho^*(P)] = 1 \cdot \frac{1}{3} + 2 \cdot \frac{8}{21} + 3 \cdot \frac{16}{105} + \dots \approx 2.21$$

- c. Argue that the following string-matching algorithm correctly finds all occurrences of pattern P in a text $T[1 \dots n]$ in time $O(\rho^*(P)n + m)$:

```
REPETITION-MATCHER(P, T)
1  m = P.length
2  n = T.length
3  k = 1 + \rho^*(P)
4  q = 0
5  s = 0
6  while s <= n - m
7      if T[s + q + 1] == P[q + 1]
8          q = q + 1
9          if q == m
10             print "Pattern occurs with shift" s
11         if q == m or T[s + q + 1] != P[q + 1]
12             s = s + max(1, ceil(q/k))
13             q = 0
```

This algorithm is due to Galil and Seiferas. By extending these ideas greatly, they obtained a linear-time string-matching algorithm that uses only $O(1)$ storage beyond what is required for P and T .

33 Computational Geometry

- 33.1 Line-segment properties
- 33.2 Determining whether any pair of segments intersects
- 33.3 Finding the convex hull
- 33.4 Finding the closest pair of points
- Problems

33.1 Line-segment properties

33.1-1

Prove that if $p_1 \times p_2$ is positive, then vector p_1 is clockwise from vector p_2 with respect to the origin $(0, 0)$ and that if this cross product is negative, then p_1 is counterclockwise from p_2 .

...

33.1-2

Professor van Pelt proposes that only the x -dimension needs to be tested in line 1 of ON SEGMENT. Show why the professor is wrong.

$(0, 0), (5, 5), (10, 0)$

33.1-3

The **polar angle** of a point p_1 with respect to an origin point p_0 is the angle of the vector $p_1 - p_0$ in the usual polar coordinate system. For example, the polar angle of $(3, 5)$ with respect to $(2, 4)$ is the angle of the vector $(1, 1)$, which is 45 degrees or $\pi/4$ radians. The polar angle of $(3, 3)$ with respect to $(2, 4)$ is the angle of the vector $(1, 1)$, which is 315 degrees or $7\pi/4$ radians. Write pseudocode to sort a sequence $\langle p_1, p_2, \dots, p_n \rangle$ of n points according to their polar angles with respect to a given origin point p_0 . Your procedure should take $O(n \lg n)$ time and use cross products to compare angles.

```
import math

def sort_by_polar_angle(p0, p):
    a = []
    for i in xrange(len(p)):
        a.append(math.atan2(p[i][1] - p0[1], p[i][0] - p0[0]))
    a = map(lambda x: x % (math.pi * 2), a)
    p = sorted(zip(a, p))
    return map(lambda x: x[1], p)
```

33.1-4

Show how to determine in $O(n^2 \lg n)$ time whether any three points in a set of n points are colinear.

Based on exercise 33.1-3, for each point p_i , let p_i be p_0 and sort other points according to their polar angles mod π . Then scan linearly to see whether two points have the same polar angle. $O(n \cdot n \lg n) = O(n^2 \lg n)$.

33.1-5

A **polygon** is a piecewise-linear, closed curve in the plane. That is, it is a curve ending on itself that is formed by a sequence of straight-line segments, called the **sides** of the polygon. A point joining two consecutive sides is a **vertex** of the polygon. If the polygon is **simple**, as we shall generally assume, it does not cross itself. The set of points in the plane enclosed by a simple polygon forms the **interior** of the polygon, the set of points on the polygon itself forms its **boundary**, and the set of points surrounding the polygon forms its **exterior**. A simple polygon is convex if, given any two points on its boundary or in its interior, all points on the line segment drawn between them are contained in the polygon's boundary or interior. A vertex of a convex polygon cannot be expressed as a convex combination of any two distinct points on the boundary or in the interior of the polygon.

Professor Amundsen proposes the following method to determine whether a sequence $\langle p_1, p_2, \dots, p_{n-1} \rangle$ of n points forms the consecutive vertices of a convex polygon.

Output "yes" if the set $\{\angle p_i p_{i+1} p_{i+2} : i = 0, 1, \dots, n - 1\}$, where subscript addition is performed modulo n , does not contain both left turns and right turns; otherwise, output "no." Show that although this method runs in linear time, it does not always produce the correct answer. Modify the professor's method so that it always produces the correct answer in linear time.

A line.

33.1-6

Given a point $p_0 = (x_0, y_0)$, the **right horizontal ray** from p_0 is the set of points $\{p_i = (x_i, y_i) : x_i \geq x_0 \text{ and } y_i = y_0\}$, that is, it is the set of points due right of p_0 along with p_0 itself. Show how to determine whether a given right horizontal ray from p_0 intersects a line segment $\overline{p_1 p_2}$ in $O(1)$ time by reducing the problem to that of determining whether two line segments intersect.

$$p_1 \cdot y = p_2 \cdot y = 0 \text{ and } \max(p_1 \cdot x, p_2 \cdot x) \geq 0$$

or

$$\text{sign}(p_1 \cdot y) \neq \text{sign}(p_2 \cdot y) \text{ and } p_1 \cdot y \cdot \frac{p_1 \cdot x - p_2 \cdot x}{p_1 \cdot y - p_2 \cdot y} \geq 0$$

33.1-7

One way to determine whether a point p_0 is in the interior of a simple, but not necessarily convex, polygon P is to look at any ray from p_0 and check that the ray intersects the boundary of P an odd number of times but that p_0 itself is not on the boundary of P . Show how to compute in $\Theta(n)$ time whether a point p_0 is in the interior of an n -vertex polygon P . (Hint: Use Exercise 33.1-6. Make sure your algorithm is correct when the ray intersects the polygon boundary at a vertex and when the ray overlaps a side of the polygon.)

Based on exercise 33.1-6, use $p_i = p_0$ as p_i .

33.1-8

Show how to compute the area of an n -vertex simple, but not necessarily convex, polygon in $\Theta(n)$ time. (See Exercise 33.1-5 for definitions pertaining to polygons.)

Half of the sum of the cross products of $\{\overrightarrow{p_1 p_i}, \overrightarrow{p_1 p_{i+1}} \mid i \in [2, n-1]\}$.

33.2 Determining whether any pair of segments intersects

33.2-1

Show that a set of n line segments may contain $\Theta(n^2)$ intersections.

Star.

33.2-2

Given two segments a and b that are comparable at x , show how to determine in $O(1)$ time which of $a \succeq_x b$ or $b \succeq_x a$ holds. Assume that neither segment is vertical.

Suppose $a = \overline{(x_1, y_1)(x_2, y_2)}$ and $b = \overline{(x_3, y_3)(x_4, y_4)}$,

$$\frac{x - x_1}{x_2 - x_1} = \frac{y - y_1}{y_2 - y_1}$$

$$y = (x - x_1) \cdot \frac{y_2 - y_1}{x_2 - x_1} + y_1$$

$$y' = (x - x_3) \cdot \frac{y_4 - y_3}{x_4 - x_3} + y_3$$

Compare y and y' . To avoid division, compare $(x_2 - x_1) \cdot y$ and $(x_4 - x_3) \cdot y'$.

33.2-3

Professor Mason suggests that we modify ANY-SEGMENTS-INTERSECT so that instead of returning upon finding an intersection, it prints the segments that intersect and continues on to the next iteration of the **for** loop. The professor calls the resulting procedure PRINT-INTERSECTING-SEGMENTS and claims that it prints all intersections, from left to right, as they occur in the set of line segments. Professor Dixon disagrees, claiming that Professor Mason's idea is incorrect. Which professor is right? Will PRINT-INTERSECTING-SEGMENTS always find the leftmost intersection first? Will it always find all the intersections?

No.

No.

33.2-4

Give an $O(n \lg n)$ -time algorithm to determine whether an n -vertex polygon is simple.

Same as ANY-SEGMENTS-INTERSECT.

33.2-5

Give an $O(n \lg n)$ -time algorithm to determine whether two simple polygons with a total of n vertices intersect.

Same as ANY-SEGMENTS-INTERSECT.

33.2-6

A **disk** consists of a circle plus its interior and is represented by its center point and radius. Two disks intersect if they have any point in common. Give an $O(n \lg n)$ -time algorithm to determine whether any two disks in a set of n intersect.

Same as ANY-SEGMENTS-INTERSECT.

33.2-7

Given a set of n line segments containing a total of k intersections, show how to output all k intersections in $O((n + k) \lg k)$ time.

Treat the intersection points as event points.

33.2-8

Argue that ANY-SEGMENTS-INTERSECT works correctly even if three or more segments intersect at the same point.

33.2-9

Show that ANY-SEGMENTS-INTERSECT works correctly in the presence of vertical segments if we treat the bottom endpoint of a vertical segment as if it were a left endpoint and the top endpoint as if it were a right endpoint. How does your answer to Exercise 33.2-2 change if we allow vertical segments?

33.3 Finding the convex hull

33.4 Finding the closest pair of points

Problems

34 NP-Completeness

- 34.1 Polynomial time
- 34.2 Polynomial-time verification
- 34.3 NP-completeness and reducibility
- 34.4 NP-completeness proofs
- 34.5 NP-complete problems
- Problems

34.1 Polynomial time

34.2 Polynomial-time verification

34.3 NP-completeness and reducibility

34.4 NP-completeness proofs

34.5 NP-complete problems

Problems

35 Approximation Algorithms

- 35.1 The vertex-cover problem
- 35.2 The traveling-salesman problems
- 35.3 The set-covering problem
- 35.4 Randomization and linear programming
- 35.5 The subset-sum problem
- Problem

35.1 The vertex-cover problem

35.2 The traveling-salesman problems

35.3 The set-covering problem

35.4 Randomization and linear programming

35.5 The subset-sum problem

Problem