Algorithms and Analysis

A short textbook for COMP 550 Midterm 1 · Arrays, Graphs, and Greedy Algorithms · UNC Fall 2026

Preface

This book covers the first three chapters of COMP 550: algorithms on arrays, the essential graph algorithms, and greedy algorithms with their correctness proofs. It is written to be read from front to back in two or three sittings. Each chapter introduces a small number of algorithms, states precisely what they compute, proves that they are correct, and analyzes how long they take. Worked examples show the algorithms running on concrete inputs, and every chapter ends with exercises whose solutions are hidden until you click. The problem-set problems from the course are included as worked examples in the chapters where they belong, because the exam reuses them.

About the exam

Midterm 1 is on Tuesday, September 22, in class. It lasts 65 minutes, is closed book with no electronics, and has assigned seating; bring your One Card. There are five problems worth five points each, and only writing inside the answer boxes is graded.

Problem 1 consists of five one-point questions in which you run an algorithm by hand on a small input and write the output with no explanation: what Two Sum returns, the distance array from BFS, the post array from DFS, which edges are back edges, and so on. Problems 2 through 5 each ask you to describe an algorithm with a given running time, explain why it is correct, and analyze its running time. At least two of these four are copied word for word from the problem sets; on the practice midterm, Problem 2 was PS 1.4 and Problem 3 was PS 2.3, and the remaining two were close variants of problem-set problems.

Conventions in force on the exam and throughout this book. Array indices start at 1. Algorithms are written in pseudocode; hashing is not allowed, so a set of numbers from 1 to n is represented by a 0/1 array of length n. Graphs are given as adjacency lists with n vertices and m edges. The following algorithms may be called by name and assumed correct: Max-in-Array, Two-Sum, Binary-Search, Selection-Sort, Merge-Sort, BFS, DFS, Explore, Find-Cycle, Topological-Sort, Prim, Kruskal, Reverse-Delete, Select-Intervals, and Fractional-Knapsack.

How solutions are written

Every problem in this course is posed as an input, a goal, and a target running time, and every solution has three parts. The *algorithm* is given in a sentence or two of English followed by short pseudocode that calls known algorithms by name. The *correctness* argument explains why the output is always what the problem asks for; this book uses four styles of argument, each introduced where it is first needed: the loop invariant (Chapter 2), the two-directional argument "anything returned is valid, and if a solution exists one is returned" (Chapter 2 and Chapter 4), case analysis on the order in which a search visits vertices (Chapter 4), and the exchange argument (Chapter 5). The *running time* is almost always the number of iterations multiplied by the cost of one iteration, plus the cost of any sorting or searching the algorithm calls; for graph algorithms the standard sentence is that each adjacency list is scanned at most once, so the total is O(m + n).

How to read this book

Read Chapters 1 and 2 first and trace the array algorithms by hand on small inputs. Chapters 3 and 4 cover graphs; draw a few small directed graphs of your own and run BFS and DFS on them until the bookkeeping is automatic. Chapter 5 is the greedy chapter, where the single most important idea is the exchange argument. Chapter 6 works the practice midterm, Chapter 7 is a second practice exam in the same format, and Chapter 8 is a one-page summary for the night before.

Blue boxes contain definitions and theorems to know by heart. Orange boxes describe mistakes that are easy to make and cost points. Green boxes connect a topic to how it is tested.

Ch 1 1. Optimization problems and the greedy idea

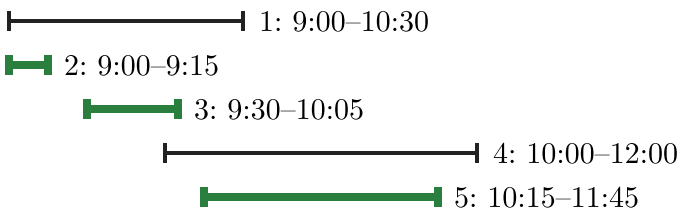
Most of the problems in this course are optimization problems: among a finite set of candidate solutions, find the best one. Two words recur in every such problem and deserve precise definitions.

Definition 1.1. A *feasible* solution is one that satisfies the constraints of the problem. An *optimal* solution, written OPT, is a feasible solution whose objective value is best possible. There may be several optimal solutions; OPT denotes any one of them. The solution produced by an algorithm is written ALG.

For example, if the problem is to attend as many events as possible from a schedule, a feasible solution is any set of events no two of which overlap in time, and an optimal solution is a feasible set of maximum size. An algorithm for an optimization problem is correct if ALG is always optimal.

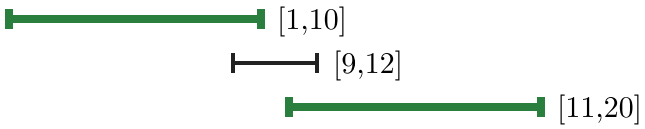
1.1 A first look at greedy algorithms

A *greedy* algorithm constructs a solution one decision at a time, choosing at each step the option that looks best by some simple rule, and never revisiting a decision. Greedy algorithms are attractive because they are simple and fast. The difficulty is that a rule which sounds sensible can be wrong, and the only way to know is to prove it right or to exhibit an input on which it fails. Consider the following schedule of five events.



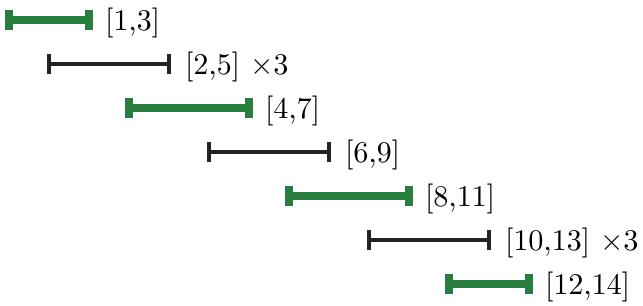
The largest set of pairwise compatible events has three members, namely events 2, 3, and 5. Suppose we try to find such a set greedily by repeatedly adding an event that does not conflict with those already chosen. Four rules suggest themselves.

Rule A: always add the shortest remaining event. This fails on the three events [1,10], [9,12], [11,20]. The shortest is [9,12], but it overlaps both of the others, so the rule chooses one event while two are possible.



Rule B: always add the event that starts earliest. This fails whenever one long event starts first and overlaps two short events that follow it: the rule takes the long event and gets one, while the two short events give two.

Rule C: always add the event that conflicts with the fewest remaining events. This rule is more subtle, and it takes eleven events to break it: [1,3], three copies of [2,5], [4,7], [6,9], [8,11], three copies of [10,13], and [12,14]. The event [6,9] conflicts with only two others, so the rule chooses it first; after that at most two more events fit, for a total of three. But the four events [1,3], [4,7], [8,11], [12,14] are pairwise compatible.



Rule D: always add the event that ends earliest. This rule is correct. The proof is the subject of Section 5.5, and it is a model for every correctness proof in Chapter 5.

Two lessons carry forward. First, a greedy algorithm is a claim that needs a proof. Second, when a greedy rule is wrong, a small counterexample is the cleanest way to say so, and you should be able to produce ones like Rules A and C on demand.

1.2 A preview of the knapsack problem

Suppose you have five dollars and three items for sale: the first is worth 5 and costs 2, the second is worth 2 and costs 1, and the third is worth 7 and costs 3. A feasible purchase costs at most 5, and the optimal one is the first and third items, worth 12. It is worth noticing that the second item has the best value per dollar of the three and yet does not appear in the optimal purchase. When items must be bought whole, the rule "best value per dollar first" can fail; this version of the problem is solved by dynamic programming in a later chapter and is not on this exam. When items may be bought in fractions, the same rule is correct, and that version is Section 5.6.

Exercises

1.1. Define feasible and optimal for the event-selection problem.

> A feasible solution is a set of events no two of which overlap. An optimal solution is a feasible set of maximum size.

1.2. Give an input on which Rule A (shortest first) is not optimal.

> [1,10], [9,12], [11,20]. The rule picks [9,12] and stops at one event; the optimum is two.

1.3. Give an input on which Rule C (fewest conflicts first) is not optimal.

[1,3], three copies of [2,5], [4,7], [6,9], [8,11], three copies of [10,13], [12,14]. The rule gets three events; the optimum is four.

1.4. Which rule is correct?

Rule D: repeatedly add the compatible event that ends earliest.

Ch 2 2. Algorithms on arrays

This chapter introduces five short algorithms on arrays. They are useful in their own right, but their main purpose is to establish how algorithms are described, how their running time is measured, and how their correctness is argued. The habits formed here are used in every later chapter.

2.1 Notation

An array A of length n has elements A[1], A[2], …, A[n]. The notation A[i:j] denotes the subarray consisting of A[i] through A[j]. Loops are written in the form `for i = 2, ..., n:`. When an algorithm needs to remember which numbers from 1 to n it has already encountered, it uses an array `B = [0]*n` of n zeros and sets B[k] = 1 upon seeing k; this replaces the hash tables that are not permitted in this course.

2.2 Maximum of an array, and the measurement of running time

Given an array A of n distinct positive integers, we wish to return the largest. The natural algorithm keeps a running maximum.

```
Max-in-Array(A):
  m = A[1]
  for i = 2, ..., n:
    m = max(m, A[i])
  return m
```

The *running time* of an algorithm is the number of primitive operations it performs in the worst case over inputs of size n, where a primitive operation is an assignment, a comparison, an arithmetic operation, an array access, or a procedure call. Counting exactly for Max-in-Array gives an expression like $4n - 1$. Such exact counts are never the object of interest. What matters is the rate of growth as n becomes large, and this is captured by *asymptotic notation*.

Big-O in practice. To convert a running-time expression to Big-O form, replace every coefficient by 1 and then keep only the fastest-growing term. Thus $4n - 1$ becomes $O(n)$, $n(n+1)/2$ becomes $O(n^2)$, $7n^3 + 9n^2 + 12$ becomes $O(n^3)$, and any constant becomes $O(1)$. The functions that arise in this book, in increasing order of growth, are $\log n$, $\sqrt{n}$, n, $n \log n$, and n^2.

Nearly every running-time analysis in this course has the same form: the number of iterations of the main loop multiplied by the cost of one iteration, plus the cost of any sorting or searching performed beforehand. For Max-in-Array there are at most n iterations of constant cost, so the running time is $O(n)$.

Correctness is argued by a *loop invariant*, a statement that is true after every iteration. Here the invariant is that after iteration i, m equals the maximum of A[1:i]. It holds after the first iteration, since m is then max(A[1], A[2]); and if it holds before iteration i, then the assignment m = max(m, A[i])

makes it hold after. When the loop ends with i = n, the invariant says m = max(A[1:n]), which is what we wanted.

2.3 Two Sum and the two-pointer technique

Let A be an array of n distinct integers sorted in increasing order, and let t be a target. We wish to find indices $i < j$ with $A[i] + A[j] = t$, or report that no such pair exists. For instance, with A = [1, 3, 4, 5, 7, 10, 11] and t = 10, the answer is (2, 5), because 3 + 7 = 10.

Examining every pair takes $O(n^2)$ time. The sorted order permits something much better. Place one pointer at each end of the array. If the two elements sum to less than t, the left pointer must move right, since every sum involving the current left element is too small; if they sum to more than t, the right pointer must move left, by the symmetric argument.

```
Two-Sum(A, t):
  i, j = 1, n
  while i < j:
    if A[i] + A[j] = t: return (i, j)
    if A[i] + A[j] < t: i += 1
    else:               j -= 1
  return nothing
```

Running time. Each iteration moves one pointer one step toward the other. The pointers begin $n - 1$ apart, so there are at most n iterations, each of constant cost, and the running time is O(n).

Correctness. There are two things to show. If the algorithm returns a pair, that pair sums to t, because the algorithm returns only upon finding such a pair. Conversely, if a solution exists, the algorithm finds one. The reason is that a solution, if one exists, always lies within A[i:j]. This is true initially, when the range is the whole array. When $A[i] + A[j] > t$, the index j cannot belong to any solution with an index at or after i, because the array is sorted and every such sum is at least A[i] + A[j]; so the algorithm may discard j without losing the solution. The case $A[i] + A[j] < t$ is symmetric. Since the range always contains a solution and shrinks by one each iteration, the loop must find the solution before the range becomes empty.

Example (from the practice midterm). Run Two-Sum on A = [1, 3, 4, 7] with t = 10. At (1, 4) the sum is 8, which is less than 10, so i becomes 2. At (2, 4) the sum is 10, and the algorithm returns (2, 4). Note that the answer consists of indices, not values.

2.4 Binary search

Given a sorted array A and a target t, we wish to find an index m with A[m] = t or report that none exists. A linear scan takes O(n) time. Because A is sorted, comparing t with the middle element tells us which half of the array could contain t, and repeating this halves the range each time.

```
Binary-Search(A, t):
  i, j = 1, n
  while i <= j:
    m = (i + j) / 2        # rounded down
    if A[m] = t: return m
    if A[m] < t: i = m + 1
    else:        j = m - 1
  return nothing
```

The length of the range $j - i + 1$ is halved at each iteration, so after about $\log_2 n$ iterations the range is empty; each iteration costs $O(1)$, and the running time is $O(\log n)$. Correctness follows the same pattern as Two-Sum: if t is present it always lies within A[i:j], because when $A[m] < t$ nothing at or before m can equal t, and symmetrically when $A[m] > t$.

A COMMON SLIP. The loop condition is $i \le j$, not $i < j$. A range containing a single element must still be examined.

2.5 Selection sort

Selection sort builds the sorted array from left to right. In round i it finds the smallest element among positions i through n and swaps it into position i.

```
Selection-Sort(A):
  for i = 1, ..., n:
    m = i
    for j = i+1, ..., n:
      if A[j] < A[m]: m = j
    swap A[i], A[m]
```

On the input [3, 1, 4, 2] the array becomes [1, 3, 4, 2] after the first round, [1, 2, 4, 3] after the second, and [1, 2, 3, 4] after the third. The invariant is that after round i the first i positions hold the i smallest elements in sorted order. There are n rounds, each scanning up to n elements, so the running time is $O(n^2)$.

2.6 Merge sort

Merge sort is an example of *divide and conquer*: split the array into two halves, sort each half recursively, and combine the two sorted halves. Combining, or *merging*, is done with two pointers, one at the front of each half; at each step the smaller of the two front elements is moved to the output.

```
Merge-Sort(A):
  if n = 1: return A
  L = Merge-Sort(A[1 : n/2])
  R = Merge-Sort(A[n/2 + 1 : n])
  return merge(L, R)
```

For example, [4, 8, 1, 3, 2, 6, 5, 7] is split into [4, 8, 1, 3] and [2, 6, 5, 7], which sort recursively to [1, 3, 4, 8] and [2, 5, 6, 7], and merging produces [1, 2, 3, 4, 5, 6, 7, 8]. Merging two halves of total length n takes O(n) time, since each step moves one element. If T(n) denotes the running time on n elements, then $T(n) = 2T(n/2) + O(n)$, and the solution of this recurrence is $O(n \log n)$. Solving recurrences is not part of this course; it suffices to know that merge sort runs in $O(n \log n)$ time.

READING A TARGET RUNNING TIME. A target of $O(n \log n)$ usually signals "sort with merge sort, then make one pass." A target of $O(n)$ on an input that is already sorted usually signals two pointers. A target of $O(n \log m)$ signals "sort the array of length m, then binary search in it once for each of the n elements of the other array."

2.7 The container problem

Let A be an array of n positive integers, thought of as the heights of vertical bars at positions 1 through n. Choosing two bars $i < j$ forms a container that holds $(j - i) \cdot \min(A[i], A[j])$ units of water. We wish to find the pair holding the most water in $O(n)$ time. (This problem is PS 1.4 and appeared as Problem 2 of the practice midterm.)

Two pointers again suffice, this time with a different rule for which pointer to move.

```
ALG(A):
  i, j = 1, n
  best = (1, n)
  while i < j:
    if area(i, j) > area(best): best = (i, j)
    if A[i] <= A[j]: i += 1
    else:            j -= 1
  return best
```

Correctness. Suppose $A[i] \le A[j]$ at some iteration. Any pair (i, j') with $j' < j$ is narrower than (i, j) and no taller, since its height is at most $A[i]$; hence no such pair holds more water than (i, j). Therefore the best container using bar i has already been examined, and moving i to the right cannot lose the optimum. The case $A[j] < A[i]$ is symmetric. Since the optimal pair is never discarded before being examined, it is recorded in best. **Running time.** The pointers meet after at most n steps of constant cost, so $O(n)$.

2.8 Worked problems from Problem Set 1

PS 1.1. An array A of length n + 1 contains every integer from 1 to n, with exactly one integer appearing twice. Find the repeated integer in O(n) time.

The sum of 1 through n is n(n + 1)/2, so the repeated integer is sum(A) − n(n + 1)/2, computed in one O(n) pass. Alternatively, use a seen-array:

```
ALG(A):
  B = [0]*n
  for i = 1, ..., n+1:
    if B[A[i]] = 1: return A[i]
    B[A[i]] = 1
```

The second time a value is encountered, its entry in B is already 1 and it is returned. Creating B and scanning A each take O(n).

PS 1.2. A is sorted in increasing order and may contain negative numbers. Return the squares of its elements in sorted order in O(n) time. For A = [−1, 0, 1, 2] the answer is [0, 1, 1, 4].

The largest square is at one of the two ends of A. Two pointers start at the ends and the output is filled from the back.

```
ALG(A):
  i, j, k = 1, n, n
  B = [0]*n
  while i <= j:
    if A[i]^2 >= A[j]^2: B[k] = A[i]^2; i += 1
    else:                B[k] = A[j]^2; j -= 1
    k -= 1
  return B
```

At every step the larger of the two end squares is the largest square remaining, so B is filled in decreasing order from the back. There are n iterations of constant cost.

PS 1.3. A has m distinct positive integers and B has n, with m ≤ n. Return the elements common to both in O(n log m) time.

```
ALG(A, B):
  sort A
  C = []
  for b in B:
    if Binary-Search(A, b) succeeds: add b to C
  return C
```

Sorting A costs O(m log m), and the n binary searches cost O(n log m). Since m ≤ n, the total is O(n log m).

PS 1.4. The container problem of Section 2.7.

Two pointers at the ends; move the pointer at the shorter bar inward; record the best area seen. O(n).

PS 1.5. A has n distinct integers. Return the smallest difference between any two of them in O(n log n) time. For A = [5, 1, 7, 10] the answer is 2.

```
ALG(A):
  sort A
  diff = ∞
  for i = 2, ..., n:
    diff = min(diff, A[i] - A[i-1])
  return diff
```

In a sorted array the closest pair must be adjacent, since any element lying between two others is closer to each of them than they are to each other. Sorting costs O(n log n) and the scan costs O(n).

Exercises

2.1. Express in Big-O form: (a) 4n − 1; (b) n(n + 1)/2; (c) $7n^3 + 9n^2 + 12$; (d) m log m + n log m with m ≤ n.

(a) O(n). (b) $O(n^2)$. (c) $O(n^3)$. (d) O(n log m).

2.2. Trace Two-Sum on A = [1, 3, 4, 5, 7, 10, 11], t = 10.

(1,7): 12, too large, j = 6. (1,6): 11, too large, j = 5. (1,5): 8, too small, i = 2. (2,5): 10, return (2, 5).

2.3. Trace Two-Sum on A = [2, 3, 5, 8], t = 9.

(1,4): 10, j = 3. (1,3): 7, i = 2. (2,3): 8, i = 3. Now i = j and the loop ends; nothing is returned.

2.4. Trace Binary-Search on A = [2, 5, 8, 12, 16, 23, 38], t = 23, listing each middle index examined.

m = 4, where A[4] = 12 < 23, so i = 5; then m = 6, where A[6] = 23, so return 6.

2.5. What is A = [5, 2, 9, 1, 7] after rounds 1 and 2 of Selection-Sort?

After round 1: [1, 2, 9, 5, 7]. After round 2: unchanged, since 2 is already the smallest of the remaining elements.

2.6. Explain in one sentence why Two-Sum runs in O(n) time.

Each iteration moves one of the two pointers one step toward the other, and they start $n - 1$ apart.

2.7. In Two-Sum, why is it safe to decrease j when A[i] + A[j] > t?

Every remaining index is at least i, and A is sorted, so pairing j with any of them gives a sum of at least A[i] + A[j] > t; thus j belongs to no remaining solution.

Ch 3 3. Graphs and breadth-first search

A graph is a set of vertices together with a set of edges joining pairs of them. Graphs model road networks, social networks, course prerequisites, and much else, and the algorithms of this chapter and the next are the basic tools for working with them.

3.1 Representing graphs

Throughout, a graph has n vertices labelled 1 through n and m edges. In an *undirected* graph an edge is an unordered pair {u, v}; in a *directed* graph an edge is an ordered pair (u, v), drawn as an arrow from u to v. In a directed graph the *out-neighbors* of u are the vertices v with (u, v) an edge, and the *out-degree* of u is the number of them.

There are two standard ways to store a graph in memory. An *adjacency matrix* is an n × n table with G[u][v] = 1 when (u, v) is an edge and 0 otherwise. Testing whether a given pair is an edge takes O(1) time, but listing the out-neighbors of u takes O(n) time regardless of how many there are, and the matrix occupies n^2 cells even for a sparse graph. An *adjacency list* stores, for each vertex u, the list G[u] of its out-neighbors. For example, G = [[2, 3, 4], [1], [4, 2], []] describes a directed graph in which 1 has edges to 2, 3, and 4; vertex 2 has an edge to 1; vertex 3 has edges to 4 and 2; and vertex 4 has no outgoing edges. Listing the out-neighbors of u takes O(out-deg(u)) time, and testing whether (u, v) is an edge means scanning G[u], which also takes O(out-deg(u)) time.

Two facts used in every graph running-time analysis. Graphs are given as adjacency lists unless stated otherwise. The lengths of all the adjacency lists sum to m, the number of edges. Consequently any algorithm that scans each list at most once and does O(1) additional work per vertex runs in O(m + n) time, which is linear in the size of the input.

3.2 Breadth-first search

The *distance* from a vertex s to a vertex v is the smallest number of edges on any path from s to v, or ∞ if no path exists. Breadth-first search, or BFS, computes the distance from a given start vertex s to every vertex. It does so by exploring the graph in layers: first the vertices at distance 1 from s, then those at distance 2, and so on. A first-in, first-out queue produces exactly this order.

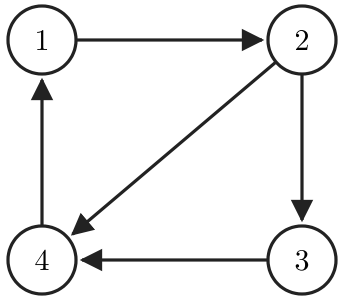
```
BFS(G, s):
  d = [∞]*n; d[s] = 0
  Q = queue containing s
  while Q is not empty:
    u = dequeue from Q
    for v in G[u]:
      if d[v] = ∞:
        d[v] = d[u] + 1
        add v to Q
  return d
```

A vertex is added to the queue the first time it is seen, and its distance is set at that moment to one more than the distance of the vertex from which it was seen. Because the queue processes vertices in order of increasing distance, the first vertex to see v is always one at minimum distance, and so d[v] is set correctly. When tracing BFS by hand, scan each adjacency list in the order written, and when two vertices could be processed in either order, take the one with the smaller label; this convention makes the result of every trace unique.

Running time. Each vertex enters the queue at most once, and processing a vertex u consists of one scan of G[u]. Summing over all vertices, the scans cost O(m) and the dequeues cost O(n), for a total of O(m + n). One could instead bound each iteration by O(n) and conclude $O(n^2)$; this is correct but loses the credit that the sharper analysis earns.

3.3 Recovering shortest paths

BFS as written reports distances but not the paths that achieve them. To recover paths, record for each vertex the vertex from which it was first seen: when d[v] is set from u, also set p[v] = u. The array p is called the *parent* array, and the edges (p[v], v) form the *BFS tree*. The shortest path from s to any reachable v is obtained by starting at v, repeatedly moving to p[v] until s is reached, and reversing the resulting list. Since each step decreases the distance by one, the walk back takes at most n steps.



In the graph above, BFS from s = 1 gives d = [0, 1, 2, 2] and p = [–, 1, 2, 2]. The path to vertex 4 is recovered as 4, p[4] = 2, p[2] = 1, which reversed is 1, 2, 4.

3.4 Worked problems from Problem Set 2

PS 2.1. Let G be a connected undirected graph. Its diameter is the largest distance between any two vertices. Compute the diameter in O(mn) time.

```
ALG(G):
  diam = 0
  for each vertex u:
    d = BFS(G, u)
    diam = max(diam, max(d))
  return diam
```

BFS from u finds the largest distance from u, and running it from every u examines every pair. Since G is connected, $m \geq n - 1$ and each BFS costs $O(m + n) = O(m)$; n calls cost $O(mn)$.

PS 2.3. Let G be an undirected graph. Return an array c such that c[u] = c[v] exactly when u and v lie in the same connected component, in O(m + n) time. (This was Problem 3 of the practice midterm.)

Run BFS from an unlabelled vertex and give every vertex it reaches the same label; then repeat from the next unlabelled vertex with a new label.

```
ALG(G):
  c = [0]*n; k = 1
  for u = 1, ..., n:
    if c[u] = 0:
      run BFS(G, u), setting c[v] = k for every vertex v it reaches
      k += 1
  return c
```

Correctness. In an undirected graph, BFS from u reaches precisely the connected component of u. Thus each component receives a single label, and since the next unlabelled vertex lies in a component not yet seen, distinct components receive distinct labels. **Running time.** Although the outer loop has n iterations, BFS is invoked only once per component, and a BFS confined to a component with a vertices and b edges costs $O(a + b)$. Summing over the components gives $O(m + n)$.

PS 2.4. A connected undirected graph is bipartite if its vertices can be colored 0 and 1 so that every edge joins vertices of different colors. Return such a coloring, or report that none exists, in O(m + n) time.

Color vertex 1 with 0 and run BFS, coloring each newly discovered vertex the opposite of the vertex that discovered it. If an edge is ever found joining two vertices of the same color, the graph is not bipartite.

```
ALG(G):
  c = [ϖ]*n; c[1] = 0
  Q = queue containing 1
  while Q is not empty:
    u = dequeue from Q
    for v in G[u]:
      if c[v] = ϖ:          c[v] = 1 - c[u]; add v to Q
      else if c[v] = c[u]: return nothing
  return c
```

Correctness. Once the color of vertex 1 is fixed, every other color is forced, since each vertex must differ from the vertex that discovered it. If the forced coloring has an edge with equal endpoints, no coloring works. If the algorithm completes, every edge {u, v} was examined when the first of its endpoints was processed and found properly colored. **Running time.** The same as BFS, O(m + n).

Exercises

3.1. Write the adjacency list of the directed graph with edges (1,3), (2,1), (2,3), (3,4), (4,2).

G = [[3], [1, 3], [4], [2]].

3.2. Run BFS from s = 2 on that graph. Give d, p, and the order in which vertices are dequeued.

Dequeue 2, discovering 1 and 3 at distance 1. Dequeue 1; its neighbor 3 is already seen. Dequeue 3, discovering 4 at distance 2. Dequeue 4; its neighbor 2 is seen. Thus d = [1, 0, 1, 2], p = [2, –, 2, 3], and the order is 2, 1, 3, 4.

3.3. Why does BFS run in O(m + n) time rather than O(n²)?

Each vertex is dequeued at most once and its list is scanned once; the lists contain m entries in total, and the dequeues cost O(n).

3.4. Modify BFS to return the shortest path from s to a given vertex t.

Record p[v] = u whenever d[v] is set. If d[t] = ∞ return nothing. Otherwise follow p from t back to s and reverse the list. The cost is O(m + n) for BFS plus O(n) for the walk.

Ch 4 4. Depth-first search and its applications

Depth-first search explores a graph by following edges as far as possible before backing up. A single depth-first search, suitably instrumented with timestamps, answers three questions about a directed graph: whether it contains a cycle, how to order its vertices so that every edge points forward, and how each edge relates to the search tree. These applications are the subject of this chapter.

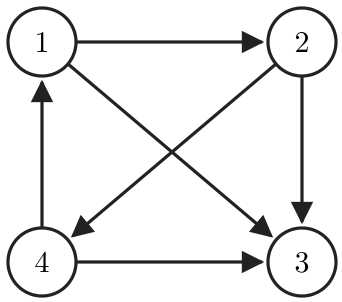
4.1 The algorithm and its timestamps

The heart of depth-first search is the recursive procedure Explore, which visits a vertex and then recursively explores each of its unvisited out-neighbors in order. A global counter t, starting at 1, stamps each vertex twice: with its *pre* value when it is first visited, and with its *post* value when Explore finishes with it. The procedure DFS calls Explore from each vertex in label order that has not yet been visited, so that every vertex is stamped even when the graph is not connected.

```
Explore(G, u):
  pre[u] = t; t += 1
  for v in G[u]:
    if pre[v] = ϖ:
      Explore(G, v)
  post[u] = t; t += 1

DFS(G):
  pre, post, t = [ϖ]*n, [ϖ]*n, 1
  for u = 1, ..., n:
    if pre[u] = ϖ:
      Explore(G, u)
  return pre, post
```

Each adjacency list is scanned exactly once, during the single call to Explore on its vertex, so DFS runs in O(m + n) time.



Example. Let G = [[2, 3], [3, 4], [], [1, 3]]. Explore(1) stamps pre[1] = 1 and, finding 2 unvisited, calls Explore(2), which stamps pre[2] = 2. The first out-neighbor of 2 is 3, so Explore(3) stamps pre[3] = 3; vertex 3 has no out-neighbors, so post[3] = 4. Returning to 2, the next out-neighbor 4 is unvisited, so

Explore(4) stamps pre[4] = 5; both out-neighbors of 4 have been visited, so post[4] = 6. Returning to 2 gives post[2] = 7, and returning to 1, whose remaining out-neighbor 3 is visited, gives post[1] = 8. Thus pre = [1, 2, 3, 5] and post = [8, 7, 4, 6].

4.2 Classification of edges

The edges along which Explore is called, here (1,2), (2,3), and (2,4), form the *DFS tree* (or forest). Every other edge falls into one of three classes according to where its head lies relative to its tail in that tree.

DEFINITION 4.1. Let (u, v) be an edge of a directed graph on which DFS has been run. The edge is a *tree edge* if Explore(v) was called from Explore(u); a *forward edge* if v is a descendant of u in the DFS tree but not by way of this edge; a *back edge* if v is an ancestor of u; and a *cross edge* if neither vertex is an ancestor of the other.

In the example, (1,3) is a forward edge, (4,1) is a back edge, and (4,3) is a cross edge. The classification can be read directly from the timestamps, without drawing the tree. Because Explore(v) begins after Explore(u) begins and ends before it ends whenever v is a descendant of u, the interval [pre[v], post[v]] is nested inside [pre[u], post[u]] for tree and forward edges; for a back edge the nesting is reversed; and for a cross edge the interval of v lies entirely before that of u.

Edge (u, v)	Timestamp pattern
tree or forward	pre[u] < pre[v] < post[v] < post[u]
back	pre[v] < pre[u] < post[u] < post[v]
cross	pre[v] < post[v] < pre[u] < post[u]

The fourth conceivable pattern, in which u finishes before v begins, cannot occur for an edge (u, v), since Explore(u) would have visited v. Tree and forward edges share a pattern; they are distinguished by whether Explore(v) was actually called from u, which is evident in a hand trace.

NOTATION. Directed edges are written with parentheses and in the direction of the arrow. A back edge from vertex 2 up to vertex 1 is written (2, 1).

4.3 Detecting cycles

THEOREM 4.2. A directed graph contains a cycle if and only if depth-first search produces a back edge.

Proof. If (u, v) is a back edge, then v is an ancestor of u, so the tree path from v down to u followed by the edge (u, v) is a cycle. Conversely, suppose the graph contains a cycle, and let v be the first vertex of the cycle that the search visits. Every other vertex of the cycle is reachable from v along the cycle

through vertices not yet visited, so all of them become descendants of v before Explore(v) finishes. In particular the vertex u preceding v on the cycle is a descendant of v, and the edge (u, v) is a back edge.

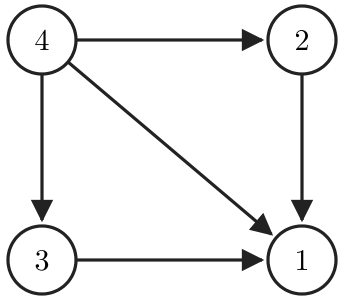
The theorem yields an algorithm that returns a cycle when one exists.

```
Find-Cycle(G):
  run DFS(G)
  for each edge (u, v):
    if pre[v] < pre[u] < post[u] < post[v]:
      return the tree path from v to u followed by (u, v)
  return nothing
```

The search costs O(m + n), the test costs O(1) per edge, and writing out the path costs O(n), so the running time is O(m + n).

4.4 Topological ordering

A *directed acyclic graph*, or DAG, is a directed graph with no cycles. A *topological ordering* of a DAG is an arrangement of its vertices in a line such that every edge points from left to right. If the vertices are courses and the edges are prerequisites, a topological ordering is a sequence in which the courses can be taken.



The DAG above has several topological orderings, among them 4, 2, 3, 1 and 4, 3, 2, 1. In general a DAG may have many; some have exactly one.

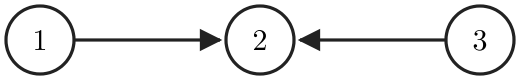
THEOREM 4.3. In any depth-first search of a DAG, post[u] > post[v] for every edge (u, v). Consequently, listing the vertices in decreasing order of post value gives a topological ordering.

Proof. Let (u, v) be an edge. If Explore(u) is called before Explore(v), then v is either visited from u or is already being explored as a descendant of u when u examines it; either way Explore(u) cannot finish until Explore(v) has finished, so post[u] > post[v]. If Explore(v) is called before Explore(u), then since the graph has no cycle there is no path from v to u, so Explore(v) finishes without ever reaching u, and post[v] < pre[u] < post[u]. In both cases post[u] > post[v].

```
Topological-Sort(G):
  R = []
  run DFS(G), and each time a vertex receives its post value, insert it at the front of R
  return R
```

The running time is that of DFS, O(m + n).

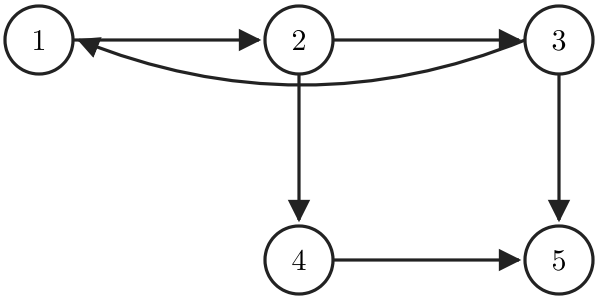
INCREASING PRE ORDER IS NOT A TOPOLOGICAL ORDERING. Consider the edges 1 → 2 and 3 → 2. The search visits 1, then 2, then 3, so increasing pre order is 1, 2, 3, but the edge 3 → 2 points backward in that order. The post values are 4, 3, 6 for vertices 1, 2, 3, and decreasing post order gives 3, 1, 2, which is valid.



A great deal follows from examining consecutive vertices in a topological ordering T. If every consecutive pair (T[i], T[i+1]) is an edge, then T is a path through all the vertices, and it is the only topological ordering, since the path fixes the relative order of every pair. If some consecutive pair is not an edge, then those two vertices cannot reach one another, and exchanging them produces a second topological ordering. Three of the problems in this chapter and one on the practice midterm are instances of this observation.

4.5 Strongly connected components

In a directed graph, two vertices are *strongly connected* if each can reach the other. A *strongly connected component* is a maximal set of mutually reachable vertices, and every directed graph partitions into such components. They can be labelled in O(m + n) time as follows: reverse every edge to obtain G^R, run DFS on G^R, and then, taking the vertices in decreasing order of post value, run BFS in the original graph from each vertex not yet labelled, giving everything it reaches a new label.



In the graph above the components are {1, 2, 3}, {4}, and {5}, and the algorithm labels them c = [3, 3, 3, 2, 1]. This algorithm is not among those that may be called by name on the exam and no problem set uses it, so it is the least important topic of the chapter.

4.6 Worked problems from Problem Set 2

PS 2.2. Let G be a DAG. Find two vertices neither of which can reach the other, or report that no such pair exists, in O(m + n) time.

```
ALG(G):
  T = Topological-Sort(G)
  for i = 1, ..., n-1:
    if T[i+1] is not in G[T[i]]:
      return (T[i], T[i+1])
  return nothing
```

Correctness. Suppose (T[i], T[i+1]) is not an edge. Then T[i+1] cannot reach T[i], because every edge points rightward in T. Nor can T[i] reach T[i+1]: every edge leaving T[i] goes to a vertex beyond T[i+1], and no edge from there points back. If instead every consecutive pair is an edge, then any vertex reaches every later vertex along the path they form, and no qualifying pair exists. **Running time.** Topological sorting costs O(m + n), and the loop scans each adjacency list at most once, for another O(m + n).

PS 2.5. Let G be a DAG. Decide in O(m + n) time whether it has more than one topological ordering.

```
ALG(G):
  T = Topological-Sort(G)
  for i = 1, ..., n-1:
    if T[i+1] is not in G[T[i]]: return True
  return False
```

If some consecutive pair is not an edge, exchanging the two vertices yields a different valid ordering, since no edge joins them and all other relative positions are unchanged. If every consecutive pair is an edge, the resulting path forces this ordering and no other. The running time is as in PS 2.2.

Exercises

4.1. Run DFS on G = [[2, 4], [3], [], [3]]. Give pre, post, and the class of every edge.

Vertex 1 is visited (pre 1), then 2 (pre 2), then 3 (pre 3, post 4); post[2] = 5; then 4 (pre 6), whose out-neighbor 3 is finished, so post[4] = 7; post[1] = 8. Thus pre = [1, 2, 3, 6] and post = [8, 5, 4, 7]. Tree edges: (1,2), (2,3), (1,4). Cross edge: (4,3).

4.2. Given pre = [1, 4, 5, 2, 3] and post = [10, 7, 6, 9, 8], classify the edges (2,1), (1,4), (3,5), (4,5).

(2,1): $1 < 4 < 7 < 10$, back. (1,4): $1 < 2 < 9 < 10$, tree. (3,5): $3 < 5 < 6 < 8$, back. (4,5): $2 < 3 < 8 < 9$, tree.

4.3. State the claim underlying Theorem 4.3 and the two cases of its proof.

For every edge (u, v), post[u] > post[v]. If u is visited first, Explore(u) cannot finish until v is finished. If v is visited first, the absence of cycles means v cannot reach u, so v finishes before u is visited.

4.4. Why does Find-Cycle examine only back edges?

Forward and cross edges lead to vertices that are not ancestors and so close no cycle, whereas a back edge together with the tree path to its tail is a cycle; and by Theorem 4.2 every cycle produces a back edge.

4.5. What ordering does Topological-Sort produce for G = [[], [1], [1, 2]]?

Vertex 1: pre 1, post 2. Vertex 2: pre 3, post 4. Vertex 3: pre 5, post 6. Decreasing post gives [3, 2, 1], which is the unique ordering since (3,2) and (2,1) are both edges.

Ch 5 5. Greedy algorithms

A greedy algorithm builds a solution by a sequence of choices, each made to look best at the moment it is made and never reconsidered. Chapter 1 showed that such a rule may or may not be optimal. This chapter presents several greedy algorithms that are optimal, and a single method, the *exchange argument*, by which each is proved so.

5.1 The exchange argument

The exchange argument establishes that a greedy algorithm's output ALG is optimal by comparing it with an optimal solution OPT and showing that OPT can be transformed into ALG without ever getting worse. It proceeds in three steps.

> **The exchange argument.**
>
> 1. Suppose, for contradiction, that OPT differs from ALG.
> 2. Locate the first decision on which they differ, and construct a new solution OPT′ from OPT by exchanging OPT's choice at that point for the choice ALG made.
> 3. Show that OPT′ is feasible and either strictly better than OPT, which is a contradiction, or at least as good as OPT while agreeing with ALG on one more decision, in which case repeating the exchange eventually transforms OPT into ALG and shows ALG optimal.

Written proofs should follow these steps in order and state explicitly what is exchanged for what. The remainder of the chapter consists of examples.

5.2 The k largest elements

Given an array A of n distinct integers and an integer $k \le n$, we wish to choose k elements with the largest possible sum. The greedy algorithm sorts A in decreasing order and returns the first k elements, in $O(n \log n)$ time.

Theorem 5.1. The k largest elements of A have the maximum sum among all k-element subsets.

Proof. Suppose OPT is a k-element subset other than the k largest. Then OPT omits some element A[i] among the k largest and, having k elements, includes some element A[j] that is not among them, so $A[i] > A[j]$. Let OPT′ be OPT with A[j] removed and A[i] added. OPT′ has k elements and its sum exceeds that of OPT by $A[i] - A[j] > 0$, contradicting the optimality of OPT.

5.3 Scheduling to minimize total waiting time

Suppose n jobs with processing times A[1], …, A[n] are to be run one after another on a single machine. The waiting time of a job is the total processing time of the jobs run before it, and we wish to order the jobs so that the sum of the waiting times is as small as possible. With processing times 4, 1, 2, the order 4, 1, 2 gives waiting times 0, 4, 5 with sum 9, whereas the order 1, 2, 4 gives 0, 1, 3 with sum 4.

The greedy algorithm runs the shortest job first, that is, sorts the jobs by increasing processing time, in O(n log n) time.

Theorem 5.2. Ordering the jobs by increasing processing time minimizes the total waiting time.

Proof. Suppose OPT is an optimal order that is not sorted. Then somewhere in OPT a job i is immediately followed by a job j with A[i] > A[j]. Let OPT′ be OPT with these two jobs exchanged. Jobs before the pair and jobs after the pair have the same waiting times as before. Job j now waits A[i] less, and job i waits A[j] more, so the total changes by A[j] − A[i] < 0. Thus OPT′ is strictly better than OPT, a contradiction.

A generalization. Problem 5 of the practice midterm attaches a weight w[i] to each job and asks to minimize the weighted sum of completion times. The answer is to sort by the ratio p[i]/w[i] in increasing order, and the proof is the same adjacent exchange; it is worked in Chapter 6. Problems that ask for an ordering are nearly always solved by sorting on the right key and proving it by exchanging two adjacent out-of-order items.

5.4 Minimum spanning trees

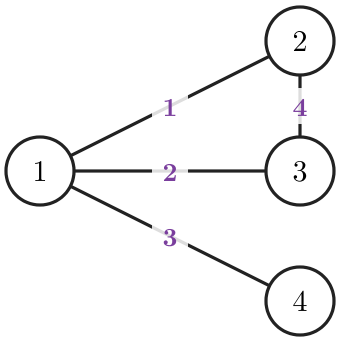
Let G be a connected undirected graph in which every edge e has a weight w(e), and assume the weights are distinct. A *spanning tree* of G is a set of n − 1 edges that connects all n vertices; equivalently, a subgraph that is connected and contains no cycle. The weight of a spanning tree is the sum of its edge weights, and a *minimum spanning tree* (MST) is a spanning tree of least weight. When the weights are distinct the MST is unique, so we may speak of *the* MST. The problem is to find it.

A *cut* is a partition of the vertices into a set S and its complement; an edge *crosses* the cut if it has one endpoint in S and one outside. The three algorithms of this section rest on two structural facts.

Theorem 5.3 (Cut property). For any cut, the lightest edge crossing it belongs to the MST.
Theorem 5.4 (Cycle property). For any cycle, the heaviest edge on it does not belong to the MST.

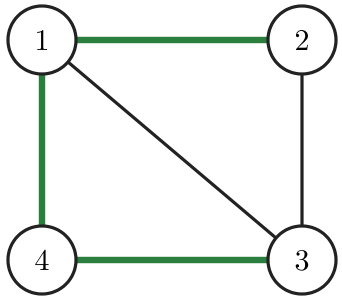
Proof of the cut property. Let e be the lightest edge crossing a cut S, and suppose e is not in the MST T. Adding e to T creates exactly one cycle, since T is a tree. This cycle crosses from S to its complement along e and must return along some other edge f, which lies in T and crosses the cut. Because e is the lightest crossing edge, w(e) < w(f). Then T + e − f is a spanning tree, since removing f breaks the only cycle without disconnecting anything, and its weight is less than that of T. This contradicts the minimality of T.

An incorrect proof. It is tempting to argue that T must contain *some* edge f crossing the cut and to exchange f for e directly. This fails because T + e − f may be disconnected when f does not lie on the cycle that e creates. For instance, let S = {1} in the graph with edges {1,2} of weight 1, {1,3} of weight 2, {1,4} of weight 3, and {2,3} of weight 4. Here e = {1,2}, and exchanging out f = {1,4} isolates vertex 4. The edge f must be chosen on the cycle.



Proof of the cycle property. Let f be the heaviest edge on a cycle C, and suppose f is in the MST T. Removing f from T splits it into two pieces, which define a cut. The cycle C crosses this cut along f and therefore crosses back along some other edge e of C, and w(e) $<$ w(f) since f is the heaviest edge of C. Then T $-$ f $+$ e is a spanning tree of smaller weight than T, a contradiction.

AN INCORRECT PROOF. One might argue that T cannot contain every edge of C and so exchange f for any edge e of C not in T. This fails because T $-$ f $+$ e may contain a cycle. Take the square with vertices 1, 2, 3, 4 in order together with the diagonal {1,3}; let C be the triangle 1, 2, 3 with f $=$ {1,2} its heaviest edge, and suppose T $=$ {1,2}, {1,4}, {3,4}. Exchanging in e $=$ {1,3} produces the cycle 1, 3, 4. The edge e must be chosen crossing the cut created by removing f.

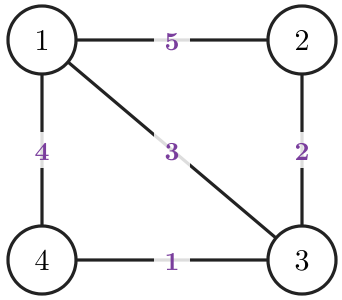


Prim's algorithm

Prim's algorithm grows a single tree outward from vertex 1. At each step it adds the lightest edge that has exactly one endpoint in the tree so far.

```
Prim(G):
  S = {1}; F = []                  # S is stored as a 0/1 array of length n
  repeat n-1 times:
    e = the lightest edge with exactly one endpoint in S
    add e to F and its outside endpoint to S
  return F
```

Each added edge is the lightest edge crossing the cut S and so lies in the MST by the cut property; after $n - 1$ additions F is the entire MST. Each of the $n - 1$ rounds scans all m edges, so the running time is $O(mn)$.



In the graph above, the edges leaving {1} have weights 5, 4, and 3, so {1,3} is added first. From {1,3} the lightest leaving edge is {3,4} of weight 1, and from {1,3,4} it is {2,3} of weight 2. The MST has weight 6.

Kruskal's algorithm

Kruskal's algorithm considers the edges in increasing order of weight and keeps each edge unless it would form a cycle with the edges already kept.

```
Kruskal(G):
  sort E by increasing weight
  F = []
  for each edge e = {u, v} in this order:
    if u cannot reach v using the edges of F:      # BFS on (V, F)
      add e to F
  return F
```

Correctness. When $e = \{u, v\}$ is added, let S be the set of vertices reachable from u using the edges of F at that moment. No edge crossing S has yet been kept, and every lighter edge has already been considered and either does not cross S or was rejected; so e is the lightest edge crossing S and belongs to the MST by the cut property. Moreover F ends as a spanning tree: if some cut had no kept edge crossing it, then any edge crossing that cut would have been kept when considered. **Running time.** Sorting costs $O(m \log n)$. Each of the m reachability tests is a BFS on the forest (V, F), which has at most $n - 1$ edges and so costs $O(n)$. The total is $O(m \log n + mn) = O(mn)$.

Reverse-Delete

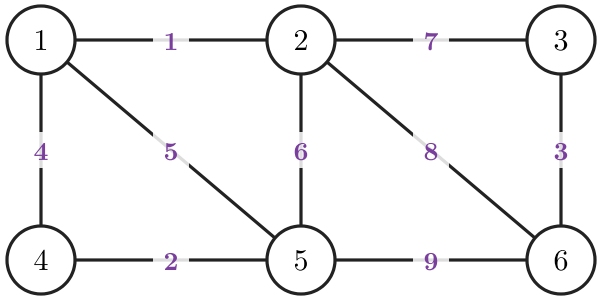
Reverse-Delete runs Kruskal's algorithm backward: it considers edges from heaviest to lightest and deletes each one unless doing so would disconnect the graph.

```
Reverse-Delete(G):
  sort E by decreasing weight
  for each edge e in this order:
    if G - e is connected:                          # BFS
      remove e from G
  return G
```

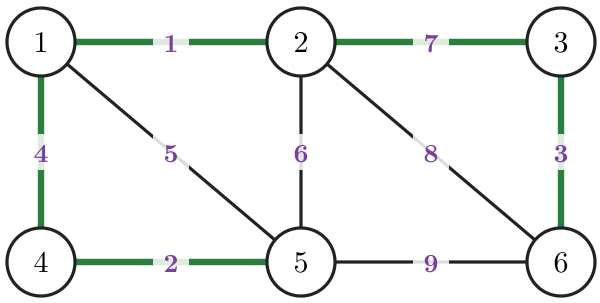
Correctness. An edge whose removal leaves the graph connected lies on a cycle, and since edges are considered heaviest first, it is the heaviest edge on that cycle; by the cycle property it is not in the MST, so the MST survives every deletion. At the end no cycle remains, for the heaviest edge of any surviving cycle could have been removed without disconnecting the graph. A connected graph without cycles that contains the MST is the MST. **Running time.** Each of the m connectivity tests is a BFS costing O(m), for a total of $O(m^2)$.

Algorithm	Order of consideration	Rule	Justification	Running time
Prim	grows from vertex 1	add the lightest edge leaving the tree	cut property	O(mn)
Kruskal	lightest edge first	keep unless it closes a cycle	cut property	O(mn)
Reverse-Delete	heaviest edge first	delete unless it disconnects	cycle property	$O(m^2)$

A worked example



In increasing order the edges are {1,2}:1, {4,5}:2, {3,6}:3, {1,4}:4, {1,5}:5, {2,5}:6, {2,3}:7, {2,6}:8, {5,6}:9. Kruskal's algorithm keeps the first four; the kept edges then connect {1, 2, 4, 5} and, separately, {3, 6}. It rejects {1,5} and {2,5}, whose endpoints are already connected, keeps {2,3}, which joins the two groups, and rejects {2,6} and {5,6}. The MST has weight 17. Prim's algorithm from vertex 1 adds {1,2}, {1,4}, {4,5}, {2,3}, and {3,6} in that order and produces the same tree.



A data structure called Union-Find performs Kruskal's cycle test faster than BFS does. It was mentioned in lecture and is not examined.

5.5 Selecting compatible intervals

An interval [s, t] with s < t represents an event beginning at time s and ending at time t. Two intervals *conflict* if they share a point, endpoints included. Given n intervals, we wish to select as many as possible, no two of which conflict. This is the event-selection problem of Chapter 1, and Rule D there is the algorithm.

```
Select-Intervals(A):
  sort A by end time
  S = []
  for each interval e in this order:
    if e begins after the last interval in S ends:
      add e to S
  return S
```

An interval conflicts with some member of S exactly when it conflicts with the last member, because the members of S were added in order of end time and the last one ends latest. The test therefore costs O(1), and the running time is that of sorting, O(n log n).

Theorem 5.5. Select-Intervals returns a maximum set of pairwise compatible intervals.

Proof. List the intervals of ALG and of an optimal solution OPT in order of end time, and suppose they differ. Let i be the first position at which ALG[i] ≠ OPT[i]. The intervals ALG[1], …, ALG[i−1] and OPT[1], …, OPT[i−1] coincide, and both ALG[i] and OPT[i] are compatible with them. The algorithm chose ALG[i] as the earliest-ending interval compatible with those choices, so OPT[i] ends no earlier than ALG[i]. Let OPT′ be OPT with OPT[i] replaced by ALG[i]. The new interval ends no later than the one it replaces, so it does not conflict with OPT[i+1] or anything after it, and it is compatible with the earlier intervals; hence OPT′ is feasible, has the same size as OPT, and agrees with ALG in its first i positions. Repeating the exchange yields an optimal solution identical to ALG, so ALG is optimal.

5.6 The fractional knapsack problem

We are given n items, the i-th having value v[i] and weight w[i], and a knapsack of capacity B. We may take any fraction x[i] between 0 and 1 of item i, receiving value x[i] · v[i] and using weight x[i] · w[i]. The goal is to maximize the total value subject to the total weight not exceeding B.

The greedy algorithm considers items in decreasing order of value per unit weight and takes as much of each as fits.

```
Fractional-Knapsack(v, w, B):
  sort the items by v[i]/w[i] in decreasing order
  x = [0]*n
  for each item i in this order:
    x[i] = min(1, remaining capacity / w[i])
  return x
```

At most one item is taken fractionally, namely the one that fills the knapsack. The running time is O(n log n) for sorting plus O(n) for the pass.

Example. Let v = [4, 1, 2], w = [5, 1, 4], and B = 7. The ratios are 0.8, 1, and 0.5, so the items are considered in the order 2, 1, 3. All of item 2 is taken (weight 1), then all of item 1 (weight 6 in total), then one quarter of item 3 to fill the remaining unit. The solution is x = [1, 1, 1/4] with value 4 + 1 + 0.5 = 5.5.

Theorem 5.6. Fractional-Knapsack returns a solution of maximum value.

Proof. Number the items so that v[1]/w[1] ≥ v[2]/w[2] ≥ . Suppose an optimal solution OPT has greater value than ALG, and let i be the first item on which they differ. The algorithm took as much of item i as would fit, so OPT takes less: OPT[i] < ALG[i]. Since OPT has at least as much value as ALG yet takes less of the early items, it must take a positive amount of some later item j > i, whose ratio is no larger. Form OPT′ by increasing x[i] by a small amount ε and decreasing x[j] by ε · w[i]/w[j]. The total weight is unchanged, so OPT′ is feasible for ε small enough. The value changes by ε · v[i] − ε · (w[i]/w[j]) · v[j] = ε · w[i] · (v[i]/w[i] − v[j]/w[j]) ≥ 0. Thus OPT′ is at least as valuable as OPT and agrees with ALG on more of item i. Repeating the exchange transforms OPT into ALG without decreasing value, so ALG is optimal.

Indivisible items. When items must be taken whole, the same greedy rule can fail. With v = [4, 4, 7], w = [5, 5, 6], and B = 10, the best ratio belongs to item 3, which fills the knapsack by itself for value 7, whereas items 1 and 2 together are worth 8. The fractional structure is essential to the exchange in the proof.

Exercises

5.1. State the three steps of an exchange argument.

Assume OPT ≠ ALG; at the first disagreement, exchange ALG's choice into OPT to obtain OPT′; show OPT′ is feasible and either strictly better than OPT (a contradiction) or as good and closer to ALG (then repeat).

5.2. State the cut property and the cycle property, and name the algorithm each justifies.

Cut property: the lightest edge crossing any cut is in the MST; it justifies Prim and Kruskal. Cycle property: the heaviest edge on any cycle is not in the MST; it justifies Reverse-Delete.

5.3. Why does Kruskal's cycle test cost O(n) rather than O(m + n)?

The search runs on the forest of kept edges, which has at most $n - 1$ edges.

5.4. Run Reverse-Delete on the graph of the worked example, listing the edges deleted in order.

{5,6} and {2,6} are deleted. {2,3} is kept, since it is now the only connection to {3, 6}. {2,5} and {1,5} are deleted. {1,4}, {3,6}, {4,5}, and {1,2} are kept. The result is the same MST of weight 17.

5.5. Run Select-Intervals on [1,4], [3,5], [0,6], [5,7], [3,9], [5,9], [6,10], [8,11], [8,12], [2,14], [12,16].

The list is already in order of end time. Select [1,4]; reject [3,5] and [0,6]; select [5,7]; reject [3,9], [5,9], [6,10]; select [8,11]; reject [8,12], [2,14]; select [12,16]. Four intervals.

5.6. Solve the fractional knapsack instance v = [10, 6, 9], w = [4, 3, 6], B = 8.

The ratios are 2.5, 2, 1.5. Take all of item 1 (weight 4) and item 2 (weight 7), then one sixth of item 3. x = [1, 1, 1/6], value 17.5.

5.7. What is the key inequality in the proof of Theorem 5.5, and why does it hold?

OPT[i] ends no earlier than ALG[i], because ALG[i] is the earliest-ending interval compatible with the shared first $i - 1$ selections, and OPT[i] is one such interval.

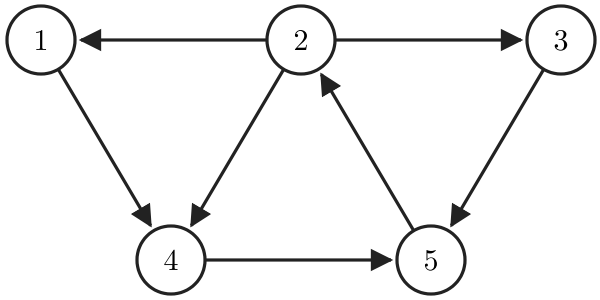
5.8. Give an instance of the indivisible knapsack problem on which the ratio rule fails.

v = [4, 4, 7], w = [5, 5, 6], B = 10: the rule obtains 7, while 8 is possible.

6. The practice midterm, with solutions

This chapter works the official practice midterm, which carries 25 points and allows 65 minutes. Attempt each problem before revealing its solution.

Problem 1 (five parts, one point each, no explanation)



The graph has adjacency list G = [[4], [1, 3, 4], [5], [5], [2]].

(a) Run Two-Sum on A = [1, 3, 4, 7], t = 10. What is returned?

At (1, 4) the sum is 8, so i becomes 2; at (2, 4) the sum is 10. The answer is **(2, 4)**.

(b) Run BFS from vertex 1. What is the array d?

From 1 we reach 4 at distance 1, then 5 at distance 2, then 2 at distance 3, then 3 at distance 4. **d = [0, 3, 4, 1, 2]**.

(c) Run DFS beginning at vertex 1. What is the array post?

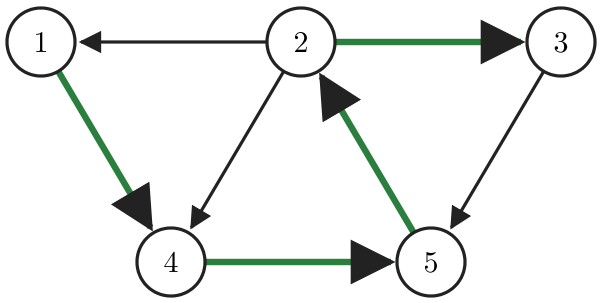
The search visits 1, 4, 5, 2 in turn (pre values 1, 2, 3, 4). From 2, the out-neighbor 1 is visited, so 3 is explored next (pre 5); its out-neighbor 5 is visited, so post[3] = 6. Back at 2, the out-neighbor 4 is visited, so post[2] = 7; then post[5] = 8, post[4] = 9, post[1] = 10. **post = [10, 7, 6, 9, 8]**, with pre = [1, 4, 5, 2, 3].

(d) List the back edges.

The tree edges are (1,4), (4,5), (5,2), (2,3). Each of the remaining edges (2,1), (2,4), and (3,5) leads to an ancestor of its tail, so all three are back edges: **(2,1), (2,4), (3,5)**.

(e) List the cross edges.

None. The DFS tree is the single path $1 \to 4 \to 5 \to 2 \to 3$, so every non-tree edge points to an ancestor.



Problem 2. The container problem (PS 1.4)

Given an array A of n positive integers, find i < j maximizing (j − i) · min(A[i], A[j]) in O(n) time.

```
ALG(A):
  i, j = 1, n; best = (1, n)
  while i < j:
    if area(i, j) > area(best): best = (i, j)
    if A[i] <= A[j]: i += 1
    else:            j -= 1
  return best
```

Correctness. When A[i] ≤ A[j], every pair (i, j′) with j′ < j is narrower than (i, j) and no taller, so the best container using bar i has already been examined and i may be advanced. The symmetric statement holds for j. The optimal pair is therefore examined before either of its bars is discarded.
Running time. The pointers meet after at most n iterations of constant cost, so O(n).

Problem 3. Connected component labels (PS 2.3)

Given an undirected graph, return an array c with c[u] = c[v] exactly when u and v are in the same component, in O(m + n) time.

```
ALG(G):
  c = [0]*n; k = 1
  for u = 1, ..., n:
    if c[u] = 0:
      run BFS(G, u), setting c[v] = k for every vertex v reached
      k += 1
  return c
```

Correctness. BFS from u reaches exactly the component of u, so each component receives one label, and the next unlabelled vertex lies in a fresh component. **Running time.** BFS runs once per component, at cost O(a + b) for a component with a vertices and b edges; summed over the components this is O(m + n).

Problem 4. A path through every vertex of a DAG

Given a DAG, return a path visiting every vertex exactly once, or report that none exists, in O(m + n) time. For G = [[2, 3], [], [2]] the answer is [1, 3, 2].

This is the consecutive-pairs observation of Section 4.4.

```
ALG(G):
  R = Topological-Sort(G)
  for i = 1, ..., n-1:
    if R[i+1] is not in G[R[i]]: return nothing
  return R
```

Correctness. If every consecutive pair of R is an edge, R is the required path. Otherwise suppose a path P through every vertex existed. P is itself a topological ordering, and it is the only one, since any other ordering would reverse some consecutive pair of P against the edge joining them. Hence R = P, and every consecutive pair of R would be an edge, contrary to what was found. **Running time.** Topological sorting costs O(m + n), and the checks scan each adjacency list at most once, another O(m + n).

For the example, DFS gives post values 6, 3, 5 for vertices 1, 2, 3, so R = [1, 3, 2]; the pairs (1,3) and (3,2) are edges, and R is returned.

Problem 5. Weighted completion time

Each element of A is a pair (w[i], p[i]). In a given ordering, C_i denotes the sum of the p-values of job i and all jobs before it. Order A to minimize Σ w[i] · C_i in O(n log n) time. For A = [(3,4), (5,1), (3,2)] in the given order the value is $3 \cdot 4 + 5 \cdot 5 + 3 \cdot 7 = 58$.

Regard p[i] as the length of job i and w[i] as the cost per unit time of delaying its completion. This is the scheduling problem of Section 5.3 with weights.

Algorithm. Sort the jobs by p[i]/w[i] in increasing order; O(n log n).

Correctness. Suppose an optimal order places job i immediately before job j although p[i]/w[i] > p[j]/w[j], that is, $p[i] \cdot w[j] > p[j] \cdot w[i]$. Exchange the two jobs. Every other job keeps its completion time. Job j now completes p[i] earlier, reducing the objective by $w[j] \cdot p[i]$; job i completes p[j] later, increasing it by $w[i] \cdot p[j]$. The net change is $w[i] \cdot p[j] - w[j] \cdot p[i] < 0$, so the exchanged order is strictly better, contradicting optimality. Hence the optimal order is sorted by p/w.

For the example the ratios are 4/3, 1/5, and 2/3, giving the order (5,1), (3,2), (3,4) with value $5 \cdot 1 + 3 \cdot 3 + 3 \cdot 7 = 35$.

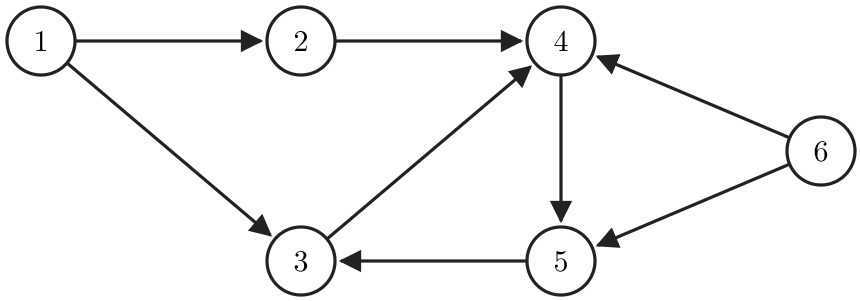
What the practice midterm suggests. Problem 1 consisted of one array trace and four facts read off a BFS or DFS, so fluent hand-tracing is worth a fifth of the exam. Two of the four design problems were problem-set problems verbatim. The remaining two were the consecutive-pairs pattern for DAGs and a sort-then-exchange greedy argument.

7. A second practice examination

The following examination follows the format of the midterm. Allow 65 minutes with the solutions hidden. For each design problem, write the algorithm, the correctness argument, and the running-time analysis as three labelled parts.

Part 1. Traces (one point each)

Parts (b) through (e) refer to the graph below, with adjacency list G = [[2, 3], [4], [4], [5], [3], [4, 5]].



(a) Run Two-Sum on A = [2, 4, 5, 9, 11, 14], t = 16. [Ch 2]

2 + 14 = 16 on the first comparison; the answer is **(1, 6)**.

(b) Run BFS from vertex 1 and give d. [Ch 3]

Vertices 2 and 3 are at distance 1, vertex 4 at distance 2, vertex 5 at distance 3, and vertex 6 is unreachable. **d = [0, 1, 1, 2, 3, ∞]**.

(c) Run DFS beginning at vertex 1 and give post. [Ch 4]

The search visits 1, 2, 4, 5, 3 in turn (pre 1 through 5). Vertex 3's out-neighbor 4 is visited, so post[3] = 6; then post[5] = 7, post[4] = 8, post[2] = 9. Vertex 1's out-neighbor 3 is visited, so post[1] = 10. The search restarts at 6 (pre 11), whose out-neighbors are visited, so post[6] = 12. **post = [10, 9, 6, 8, 7, 12]**.

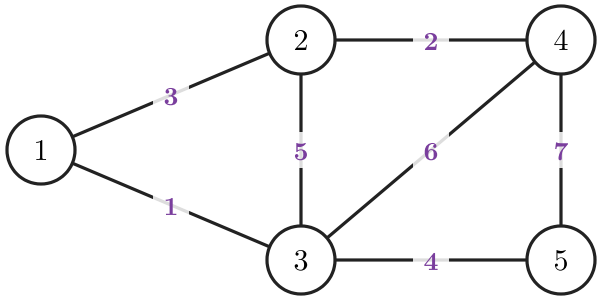
(d) List the back edges. [Ch 4]

(3, 4) only; vertex 4 is an ancestor of 3 by way of 4 → 5 → 3.

(e) List the cross edges and the forward edges. [Ch 4]

Cross edges: **(6, 4)** and **(6, 5)**, since 4 and 5 were finished before 6 was visited. Forward edge: **(1, 3)**, since 3 is a descendant of 1 reached by way of 2, 4, and 5. The tree edges are (1,2), (2,4), (4,5), (5,3).

Parts (f) through (h) refer to the weighted graph below.



(f) List the edges Kruskal's algorithm keeps, in order. [Ch 5]

In increasing order the edges are {1,3}:1, {2,4}:2, {1,2}:3, {3,5}:4, {2,3}:5, {3,4}:6, {4,5}:7. The first four are kept; the last three each close a cycle. **{1,3}, {2,4}, {1,2}, {3,5}**, of total weight 10.

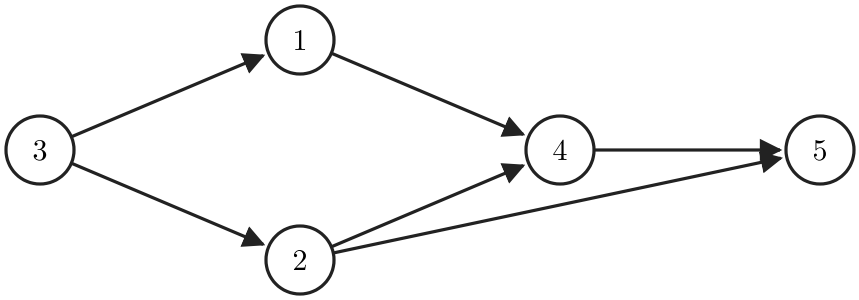
(g) List the edges Prim's algorithm adds from vertex 1, in order. [Ch 5]

{1,3} of weight 1; then, leaving {1, 3}, the lightest edge is {1,2} of weight 3; then {2,4} of weight 2; then {3,5} of weight 4. **{1,3}, {1,2}, {2,4}, {3,5}**.

(h) List the edges Reverse-Delete removes, in order. [Ch 5]

{4,5}, {3,4}, and {2,3} are removed in turn. {3,5} is kept, since its removal would isolate 5; {1,2} is kept, since its removal would separate {2, 4}; {2,4} and {1,3} are kept. **Removed: {4,5}, {3,4}, {2,3}**.

Parts (i) and (j) refer to the DAG below, with adjacency list G = [[4], [4, 5], [1, 2], [5], []].



(i) What ordering does Topological-Sort return? [Ch 4]

The search visits 1, 4, 5 (pre 1, 2, 3), giving post[5] = 4, post[4] = 5, post[1] = 6; then 2 (pre 7, post 8); then 3 (pre 9, post 10). Decreasing post order is **[3, 2, 1, 4, 5]**.

(j) Does the DAG have more than one topological ordering? Does it have a path through every vertex? [Ch 4]

The consecutive pair (2, 1) in [3, 2, 1, 4, 5] is not an edge. Hence there is a second ordering, [3, 1, 2, 4, 5], and there is no path through every vertex.

(k) Run Fractional-Knapsack on v = [6, 10, 12], w = [2, 5, 8], B = 9. [Ch 5]

The ratios are 3, 2, and 1.5. All of item 1 (weight 2) and item 2 (weight 7) are taken, then 2 of the 8 units of item 3. **x = [1, 1, 1/4]**, value 19.

(l) Run Select-Intervals on [2,5], [1,3], [4,8], [6,7], [3,4], [7,9]. [Ch 5]

In order of end time: [1,3], [3,4], [2,5], [6,7], [4,8], [7,9]. Select [1,3]; reject [3,4], which shares the point 3; reject [2,5]; select [6,7]; reject [4,8]; reject [7,9], which shares the point 7. **[1,3], [6,7]**.

Part 2. Arrays

2. A is sorted in increasing order with distinct elements, and t > 0. Find i < j with A[j] − A[i] = t, or report that no such pair exists, in O(n) time. [Ch 2]

Use two pointers that both begin at the left and move only rightward.

```
ALG(A, t):
  i, j = 1, 2
  while j <= n:
    if A[j] - A[i] = t: return (i, j)
    if A[j] - A[i] < t: j += 1
    else:               i += 1
    if i = j: j += 1
  return nothing
```

Correctness. If the difference is less than t, no $j' \le j$ pairs with the current i, since moving j leftward only decreases the difference; so j may advance. If the difference exceeds t, no $i' \le i$ pairs with the current j, so i may advance. No solution is ever skipped, and any returned pair is verified directly. **Running time.** Each iteration advances one pointer, and neither exceeds n, so there are at most 2n iterations: O(n).

2′. A of length m and B of length n are each sorted. Produce a single sorted array of all m + n elements in O(m + n) time. [Ch 2]

This is the merge step of merge sort. Keep a pointer at the front of each array; repeatedly move the smaller front element to the output and advance its pointer; when one array is exhausted, copy the remainder of the other. The smaller front element is smaller than everything remaining in both arrays, so the output is sorted. Each iteration moves one element, so there are m + n iterations.

Part 3. Breadth-first search

3. G is undirected with vertices s and t. Return a shortest path from s to t as a list of vertices, or report that t is unreachable, in O(m + n) time. [Ch 3]

```
ALG(G, s, t):
  run BFS(G, s), recording p[v] = u whenever d[v] is set from u
  if d[t] = ∞: return nothing
  P = [t]
  while the last vertex of P is not s: append p[last vertex of P]
  reverse P and return it
```

Correctness. Each p[v] is a neighbor of v with d[p[v]] = d[v] − 1, so following p from t traverses a path of length d[t] to s, which is shortest. **Running time.** BFS costs O(m + n), and the walk back takes at most n steps.

3′. G is directed and s is a vertex. Decide whether every vertex can reach s, in O(m + n) time. [Ch 3]

Construct the reversed graph in O(m + n) by inserting u into R[v] for every u and every v in G[u]. Run BFS from s in the reversed graph and answer yes exactly when no entry of d is ∞. A vertex v reaches s in G exactly when s reaches v in the reversed graph.

Part 4. Depth-first search and DAGs

4. G is directed. Return a topological ordering if G is a DAG, and otherwise return a cycle, in O(m + n) time. [Ch 4]

Run DFS once. If some edge (u, v) satisfies pre[v] < pre[u] < post[u] < post[v], it is a back edge; return the tree path from v to u followed by (u, v). Otherwise return the vertices in decreasing order of post value. By Theorem 4.2 a cycle exists exactly when a back edge does, and by Theorem 4.3 the decreasing-post order is topological when no back edge exists. The cost is O(m + n) for the search plus O(m) for the edge tests.

4′. G is a DAG and s is a vertex. Count the vertices that appear after s in every topological ordering of G, in O(m + n) time. [Ch 4]

These are exactly the vertices reachable from s, so run BFS from s and count the vertices other than s with finite distance. If s reaches v, every ordering places v after s, since each edge of the path points rightward. If s does not reach v, adding the edge (v, s) creates no cycle, and a topological ordering of the enlarged DAG places v before s while remaining valid for G. One BFS suffices: O(m + n).

Part 5. Greedy algorithms

5. Given n intervals, choose the fewest points on the line such that every interval contains at least one chosen point. Give an O(n log n) algorithm with proof. [Ch 5]

```
ALG(A):
  sort A by end time
  P = []; last = -ϖ
  for each [s, t] in this order:
    if s > last:
      add t to P; last = t
  return P
```

Correctness. Call an interval a trigger if the algorithm added a point upon reaching it. Each trigger begins after the previous point, which was the end of the previous trigger, and no later interval ends earlier than the current one; hence the triggers are pairwise disjoint. Every valid set of points contains a distinct point in each trigger, so no valid set is smaller than the algorithm's, which has one point per trigger. **Running time.** Sorting costs O(n log n), and the pass costs O(n).

5′. A contains n distinct box weights and B is a truck's capacity. Load as many boxes as possible without exceeding B, in O(n log n) time, with an exchange argument. [Ch 5]

Sort the weights in increasing order and load boxes until the next does not fit; say k boxes are loaded. Suppose OPT loads at least k boxes but is not a set of the lightest ones. Then OPT omits some box among the |OPT| lightest and includes a heavier one; exchanging the lighter box in preserves the count and reduces the total weight, so feasibility is kept. Repeating shows that the |OPT| lightest boxes fit; but the algorithm loaded the greatest number of lightest boxes that fit, so |OPT| ≤ k. Sorting and the pass cost O(n log n).

5″. G is connected and undirected with distinct edge weights, and e = {u, v} is an edge. Decide in O(m + n) time whether e belongs to the MST. [Ch 5]

Discard every edge heavier than e, including e itself, and run BFS from u. If v is reached, the path found together with e forms a cycle on which e is heaviest, so e is not in the MST by the cycle property. If v is not reached, let S be the set of vertices reached; every edge leaving S other than e is heavier than e, so e is the lightest edge crossing S and lies in the MST by the cut property. Filtering and searching cost O(m + n).

Part 6. Short questions

(a) Why is O(m + n) regarded as linear time for graph algorithms?

The adjacency list itself has n lists and m entries, so O(m + n) is linear in the size of the input.

(b) Give the running time of Max-in-Array, Two-Sum, Binary-Search, Selection-Sort, Merge-Sort, BFS, DFS, Find-Cycle, Topological-Sort, Prim, Kruskal, Reverse-Delete, Select-Intervals, and Fractional-Knapsack.

O(n), O(n), O(log n), $O(n^2)$, O(n log n), O(m + n), O(m + n), O(m + n), O(m + n), O(mn), O(mn), $O(m^2)$, O(n log n), O(n log n).

(c) A DAG has exactly one topological ordering. What does this imply about its edges?

Every consecutive pair in that ordering is an edge, so the DAG contains a path through all of its vertices.

(d) Name a problem for which "best ratio first" is optimal and one for which it is not.

Optimal for fractional knapsack and for weighted completion time; not optimal for the indivisible knapsack problem, as v = [4, 4, 7], w = [5, 5, 6], B = 10 shows.

8. Summary

This chapter collects the definitions, algorithms, and facts of the book in the form of a single reference page.

Writing solutions. Three labelled parts: algorithm in pseudocode, correctness, running time. Indices begin at 1. No hashing; use a 0/1 array. Graphs are adjacency lists. Running time is iterations times cost per iteration, plus any sort or search. For graphs: each list scanned once gives $O(m + n)$.

Asymptotics. Discard constants and lower-order terms. $\log n < \sqrt{n} < n < n \log n < n^2$. Merge sort is $O(n \log n)$. In a connected graph $m \geq n - 1$, so $O(m + n) = O(m)$ and $\log m = O(\log n)$.

Arrays. Two-Sum on a sorted array: pointers at both ends; a sum too small advances i, too large retreats j; at most n steps; a solution always remains in A[i:j]. Binary-Search halves the range with condition $i \leq j$. Selection-Sort $O(n^2)$; Merge-Sort $O(n \log n)$. Container problem: advance the pointer at the shorter bar. An $O(n \log n)$ target suggests sorting then one pass; an $O(n)$ target on sorted input suggests two pointers.

Problem Set 1. 1.1 The repeated element is $\text{sum}(A) - n(n+1)/2$, or found with a seen-array. 1.2 Sorted squares: pointers from both ends, filling from the back. 1.3 Intersection: sort the shorter array and binary search for each element of the longer, $O(n \log m)$. 1.4 Container problem. 1.5 Minimum difference: sort and compare neighbors.

Breadth-first search. Queue; $d[s] = 0$; on first seeing v, set $d[v] = d[u] + 1$ and $p[v] = u$. Smaller labels first when tracing. $O(m + n)$. Shortest path: follow p from t and reverse.

Depth-first search. Stamp pre on entry and post on exit from one clock. Edge (u, v) is a back edge if $\text{pre}[v] < \text{pre}[u] < \text{post}[u] < \text{post}[v]$; a cross edge if v finished before u began; tree or forward if v's interval nests inside u's. Write directed edges as (tail, head). Cycle exists iff a back edge exists. Topological ordering: decreasing post; for each edge, $\text{post}[u] > \text{post}[v]$. Increasing pre order fails.

Problem Set 2. 2.1 Diameter: BFS from every vertex, $O(mn)$. 2.2 Mutually unreachable pair in a DAG: first consecutive non-edge in a topological ordering. 2.3 Component labels: BFS from each unlabelled vertex with a fresh label; $O(m + n)$ since one BFS per component. 2.4 Bipartite: BFS coloring $1 - c[u]$; fail on an equal-colored edge. 2.5 More than one ordering iff some consecutive pair is not an edge. Practice Problem 4: a path through all vertices iff every consecutive pair is an edge.

The exchange argument. Assume OPT $\neq$ ALG; at the first difference, exchange ALG's choice into OPT; show the result is feasible and either strictly better (contradiction) or as good and closer to ALG (repeat). k largest: exchange a small chosen element for a large omitted one. Waiting time: shortest first; an adjacent exchange saves $A[i] - A[j]$. Weighted completion: sort by p/w; an adjacent exchange saves $w[j]p[i] - w[i]p[j]$.

Minimum spanning trees. Spanning tree: $n - 1$ edges, connected, acyclic. Distinct weights give a unique MST. Cut property: the lightest edge across any cut is in the MST. Cycle property: the heaviest edge on any cycle is not. Proofs exchange along the cycle created by adding e, or across the cut created by removing f; an arbitrary exchange partner may disconnect or create a cycle. Prim grows from vertex 1 by the lightest leaving edge, $O(mn)$. Kruskal takes lightest first unless a cycle closes, $O(mn)$. Reverse-Delete removes heaviest first unless the graph disconnects, $O(m^2)$.

Intervals and knapsack. Select-Intervals: sort by end time; take an interval if it begins after the last taken one ends; $O(n \log n)$; proof: OPT[i] ends no earlier than ALG[i], exchange, repeat. Failing rules: shortest first, earliest start, fewest conflicts. Fractional-Knapsack: highest v/w first, as much as fits; proof: shift ε of weight from a lower-ratio item to a higher-ratio one. Indivisible counterexample: $v = [4,4,7]$, $w = [5,5,6]$, $B = 10$.

Practice midterm answers. 1(a) (2,4). 1(b) $d = [0,3,4,1,2]$. 1(c) post $= [10,7,6,9,8]$. 1(d) back edges (2,1), (2,4), (3,5). 1(e) none. Problem 2: container. Problem 3: component labels. Problem 4: topological ordering with consecutive edges. Problem 5: sort by p/w with adjacent exchange.

Running times. Max-in-Array $O(n)$ · Two-Sum $O(n)$ · Binary-Search $O(\log n)$ · Selection-Sort $O(n^2)$ · Merge-Sort $O(n \log n)$ · BFS, DFS, Find-Cycle, Topological-Sort $O(m + n)$ · Prim $O(mn)$ · Kruskal $O(mn)$ · Reverse-Delete $O(m^2)$ · Select-Intervals $O(n \log n)$ · Fractional-Knapsack $O(n \log n)$.

The examination. Tuesday, September 22; 65 minutes; closed book; five problems of five points; assigned seats; One Card required. Problem 1: five traces without explanation or partial credit. Problems 2–5: problem-set style, at least two taken verbatim from the problem sets. Write only inside the boxes.

Built September 19, 2026 from the COMP 550 lecture notes, Chapter 3 notes, Problem Sets 1–2, and Practice Midterm 1 (UNC Fall 2026). Personal study material.