COMP2123 Finals Notes

Induction

Induction is a very useful proof technique for proving correctness and bound the running time of an algorithm. Most of the statements about the running times of algorithms that we'll see during this course will hold *for all n> 0,* where n is the size of the input. However, this is a claim about an infinite set of numbers and hence we can't prove it for each element separately.

Base Case
Induction proves the statement for a *base case***,** which is the smallest element for which the claim must hold (n = 1).

Induction Step
Assumes that the claim holds for all values of n smaller than the current one. This is called this induction hypothesis. Using the induction hypothesis, we then show that the claim holds for the next value of n.
Ie. Using the induction hypothesis for n – 1 to prove the claim for n or using the induction hypothesis for n/2 to prove the claim for n when n is even.

Problems

Problem 1.
Recall that one way of defining the Fibonacci sequence is as follows:
$F(1)=1, F(2)=2, \wedge F(n)=F(n-2)+F(n-1) \text{ for } n>2.$
Use induction to prove that $F(n)<2^n$.

Base Case: n = 1 and n = 2. We have that $F(1)=1<2^1 \wedge F(2)=2<2^2=4.$

Induction Step: n > 2. Assume that the claim is true for n' < n. Consider n. We need to show that $F(n)<2^n$.

Since n > 2, the definition of the Fibonacci sequence gives us that
$F(n)=F(n-2)+F(n-1).$
Next, we apply the induction hypothesis on F(n – 2) and F(n – 1), which gives us
$$F(n)<2^{n-2}+2^{n-1}$$
Rearranging the equation, we show that
$F(n)<2^{n-2}+2^{n-1} \quad ¿ 2^{n-1}+2^{n-1} \quad ¿ 2\times 2^{n-1} \quad ¿ 2^n$

Problem 2. (C-1.5)
Consider the following recurrence equation, defining a function T(n):
$T(n)=\begin{cases} 1 \\ T(n-1)+n \end{cases}, \text{if } n=1. \text{Otherwise},$ Show, by induction, that $T(n)=n(n+1)/2.$

Problem 3. (C-1.7)
Consider the following recurrence equation, defining a function T(n):
$T(n)=\begin{cases} 1 \\ 2T(n-1) \end{cases}, \text{if } n=0. \text{Otherwise},$ Show, by induction, that $T(n)=2^n.$

Analysis

When analysing algorithms, we desire an analytic framework that:

- Considers all possible inputs
- Allows us to evaluate the relative efficiency of any two algorithms in a way that is independent from the hardware and software environment
- Can be performed by studying a high level description of the algorithm without actually implementing it or running experiments on it.

Algorithm Efficiency

An algorithm is efficient if it runs in polynomial time; that is, on an instance of size n, it performs no more than p(n) steps for some polynomial $p(s)=a_d x^d+\ldots+a_1 x+a_0$
This gives us some information about the expected behaviour of the algorithm and is useful for making predictions and comparing different algorithms.

Using Pseudocode

We want our pseudocode descriptions to always be detailed enough to fully justify the correctness of the algorithm they describe, while being simple enough for human readers to understand.

Correctness

We must use mathematical language to justify for prove our statements.

1. Proof by Counter Example

Typically used to prove something like "Every element x in a set S has property P".

Example: A certain Professor Amongus claims that every number of the form 2^i-1 is a prime, when i is an integer greater than 1. Prove he is wrong.

To prove him wrong, we need to find a counter example:

$$2^4-1=15=3\times 5$$

2. Contrapositives and Contradiction

Another set of justification techniques involves the use of the negative. The two primary methods are the use of the contrapositive and the contradiction. To justify the statement "if p is true, then q is true", we instead establish that "if q is not true, then p is not true."

3. Induction

Most of the claims we make about a running time or a space bound involve an integer parameter n (usually denoting the 'size' of the problem). Most of these claims are equivalent to saying some statement q(n) is true for all "n $\geq$ 1." Since this is making a claim about an infinite set of numbers, we cannot justify this exhaustively in a direct fashion.

By use of induction, we can show that for any n $\geq$ 1, there is a finite sequence of implications that starts with something known to be true and ultimately leads to showing that q(n) is true. Specifically, a base base (n = 1). Then, the inductive step for n > k.

4. Loop Invariants

To show an algorithm is correct, we can a loop invariant argument. That is, we inductively define statements, $S_i, for\, i=0,1,2\ldots,n$, that lead to the correctness of the algorithm.

Asymptomatic Notation (Running Time)

logarithmic	linear	quadratic	polynomial	exponential
$O(\log n)$	$O(n)$	$O(n^2)$	$O(n^k)\ (k \geq 1)$	$O(a^n)\ (a > 1)$

Table 1.6: Terminology for classes of functions.

In Table 1.10, we illustrate the difference in the growth rate of the functions shown in Table 1.9.

n	$\log n$	$\log^2 n$	$\sqrt{n}$	$n \log n$	n^2	n^3	2^n
4	2	4	2	8	16	64	16
16	4	16	4	64	256	4,096	65,536
64	6	36	8	384	4,096	262,144	1.84×10^{19}
256	8	64	16	2,048	65,536	16,777,216	1.15×10^{77}
1,024	10	100	32	10,240	1,048,576	1.07×10^{9}	1.79×10^{308}
4,096	12	144	64	49,152	16,777,216	6.87×10^{10}	10^{1233}
16,384	14	196	128	229,376	268,435,456	4.4×10^{12}	10^{4932}
65,536	16	256	256	1,048,576	4.29×10^{9}	2.81×10^{14}	10^{19728}
262,144	18	324	512	4,718,592	6.87×10^{10}	1.8×10^{16}	10^{78913}

Problems

Problem 1. Consider the following pseudo-code fragment.

```
def stars(n):
      for i in [1:n]:
            print "*" i many times
```

(a) Using the O-notation, upper bound the running time of stars.
The first iteration prints 1 star, second prints 2 third prints 3 and so on. The total number of stars is 1 + 2 + … + n, namely,

$$\sum_{j=1}^{n} j \leq \sum_{j=1}^{n} n = n^2 = O(n^2)$$

(b) Using the Ω -notation, lower bound the running time of stars to show that your upper bound is in fact asymptotically tight.
Assume for simplicity that n is even. To lower bound the running time, we consider only the number of stars printed during the last $\frac{n}{2}$ iterations. Since this is part of the full execution, analysing only this part gives a lower bound on the total running

time. The main observation that we need is that for each of the considered iterations, we print at least $\frac{n}{2}$ stars, allowing us to lower bound the total number of stars printed:

$$\sum_{j=1}^{n} j \leq \sum_{j=n/2+1}^{n} \frac{n}{2} = \frac{n^2}{4} = \Omega(n^2)$$

Problem 2. Given an array with *n* integer values, we would like to know if there are any duplicates in the array. Design an algorithm for this task and analyse its time complexity.

The straightforward solution is to do a double for loop over the entries of the array, returning "found duplicates" right away when we find a pair of identical elements, and "found no duplicates" at the end if we exit the for loops.

Correctness: This algorithm is correct, since we compare every pair of elements, thus if there exists a duplicate pair, we consider this pair at some point in the execution. Similarly, if no duplicates exist, we can safely return this after checking all pairs.

Complexity: The complexity of this solution is $O(n^2)$, since there are $O(n^2)$ pairs of elements to consider and performing a single such comparison take $O(1)$ time.

A better solution is to sort the elements and then do a linear time scan testing adjacent positions. The correctness follows from the fact that in a sorted array, duplicate integers must occur in consecutive indices. An array of length n can be sorted in $O(nlogn)$ time, which also dominates the running time of this algorithm, as scanning through n elements and performing a single comparison for consecutive integers takes $O(1)$ time per pair and $O(n)$ time in total.

Lists

Lists maintain linear orders while allowing for access and updates in the 'middle'.

Array-Based Lists

An obvious choice for implementing an index-based list is to use an array A, where A[i] stores (a reference to) the element with index i. We choose the size N of array A to be sufficiently large, and we maintain in an instance variable the actual number n < N of elements in the list.

Add and Remove

An important and time-consuming part of these implementations involves the shifting of elements up or down to keep the occupied cells in the array continuous. These shifting operations are required to maintain our rule of always storing an element of rank i at index i in A.

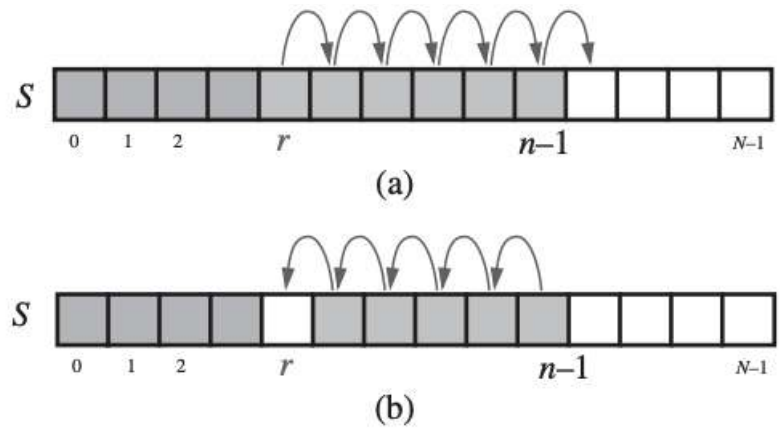


Figure 2.8: Array-based implementation of an index-based list, S: (a) shifting up for an insertion at index r; (b) shifting down for a removal at index r.

Method	Time
get(r)	$O(1)$
set(r, e)	$O(1)$
add(r, e)	$O(n)$
remove(r)	$O(n)$

Table 2.9: Worst-case performance of an index-based list with n elements implemented with an array. The space usage is $O(N)$, where N is the size of the array.

Singly Linked List

A concrete data structure which consists of a sequence of nodes, each with a reference to the next node. The list is captured by reference to the first node (the 'head'). A linked list is a container of elements that stores each element at a node position and that keeps these positions arranged in a linear order relative to one another.

A position (or node) is always defined relatively, that is, in terms of its neighbours.

Method	Time
first()	$O(1)$
last()	$O(1)$
before(p)	$O(1)$
after(p)	$O(1)$
insertBefore(p, e)	$O(1)$
insertAfter(p, e)	$O(1)$
remove(p)	$O(1)$

Table 2.15: Worst-case performance of a node-based linked list with n elements. The space usage is $O(n)$, where n is the number of elements in the list.

Array or Linked List?

Array	Linked List
- Good match to index-based abstract data type - Caching makes traversal fast in practice - No extra memory needed to store pointers - Allows random access (retrieve element by index)	- Good match to positional ADT - Efficient insert and deletion - Simpler behaviour as collection grows - Modifications can be made as collection is iterated over - Space not wasted by having maximum capacity

Problems

Problem 1. (Assignment 1)
We want to design a list-like data structure that supports the following operations on integers.

- AddToFront(e): adds a new element e to the front of the list
- AddToBack(e): adds a new element e to the back of the list
- RemoveFromFront():
- RemoveFromBack():
- SetElement(i, e):

Each element should run in O(1) time. You can assume that we know that the list will never contain more than k elements (k is not a constant and should not show up in your running time).

(a) Design a data structure that supports the above operations in the required time.
(b) Briefly argue the correctness of your data structure and operations
(c) Analyse the running time of your operations

Problem 2. (Assignment 1)
We are given $n \geq 2$ wireless sensors modelled as points on a 1D line and every point has a distinct location modelled as a positive x-coordinate. These sensors are given as a sorted array A. Each wireless sensor has the same broadcast radius r. Two wireless sensors can send messages to each other if their locations are at most distance r apart. Your task is to design an algorithm that returns all pairs of sensors that can communicate with each other, possibly by having their messages forwarded via some of the intermediate sensors. For full marks your algorithm needs to run in $O(n^2)$ time.

(a) Design an algorithm that solves the problem.
(b) Briefly argue the correctness of your algorithm.
(c) Analyse the running time of your algorithm.

Stacks and Queues

Stacks

A stack is a container of objects that are inserted and removed according to the **last-in-first-out** (LIFO) principle. Only the most-recently inserted object can be removed at any time.

A stack can be implement with an N-element array S, with elements stored from S[0] to S[t], where t is an integer that gives the index of the top element in S. We must specify some maximum size N for our stack.

Figure 2.2: Implementing a stack with an array S. The top element is in cell $S[t]$.

Algorithm push(o):
  if $t + 1 = N$ **then**
    return that a stack-full error has occurred
  $t \leftarrow t + 1$
  $S[t] \leftarrow o$
  return

Algorithm pop():
  if $t < 0$ **then**
    return that a stack-empty error has occurred
  $e \leftarrow S[t]$
  $S[t] \leftarrow$ **null**
  $t \leftarrow t - 1$
  return e

Algorithm 2.3: Implementing a stack with an array.

The above methods push and pop run in a constant amount of time. This is because they only involve a constant number of simple arithmetic operations, comparisons, and assignment statements. That is, in this array-based implementation of the stack, each method runs in O(1) time.

Queues

A queue is a container of objects that are inserted and removed according to the **first-in-first-out** (FIFO) principle. Only the element that has been in the queue the longest can be removed at any time.

An array-based implementation of a queue means we must decide how we are going to keep track of the front and rear of the queue. We define two variables *f* and *r* which have the following meanings:

- *f* is an index to the cell of Q, storing the first element of the queue (which is the next candidate to be removed by a dequeue operation), unless the queue is empty (in which case $f = r$.
- *r* is an index to the next available array cell in Q.

Initially, we assign $f = r = 0$ and we indicate that the queue is empty by the condition $f = r$. When we remove an element from the front of the queue, we can simply increment *f* to index the next cell. Likewise, when we add an element, we can increment r to index the next available cell in Q.

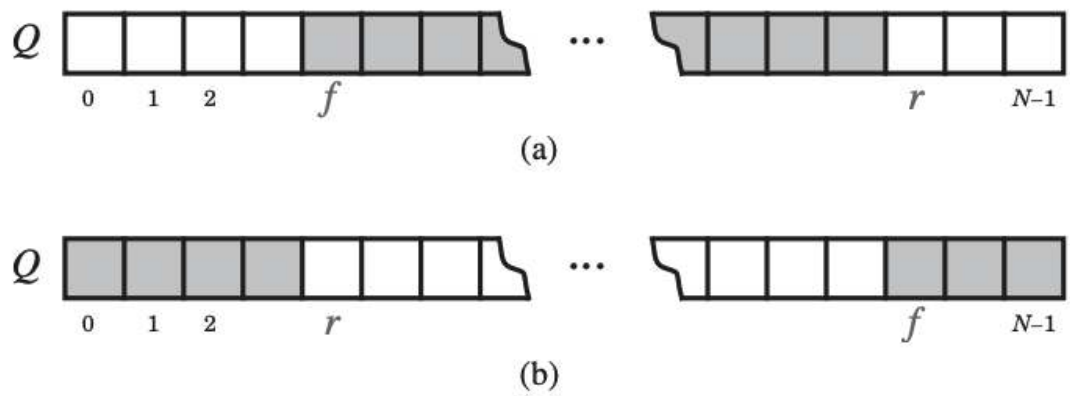


Figure 2.5: Implementing a queue using an array Q in a circular fashion: (a) the "normal" configuration with $f \leq r$; (b) the "wrapped around" configuration with $r < f$. The cells storing queue elements are highlighted.

Each method executes a constant number of statements involving arithmetic operations, comparisons and assignments. Thus, an array based implementation of a queue runs in O(1) time.

Problems

Problem 1. Given a singly linked list, we would like to traverse the elements of the list in reverse order. Design an algorithm that uses $O(\sqrt{n})$ extra space. What is the best time complexity you can get?

We store $\sqrt{n}$ cursors evenly spaced along the list. We traverse the span between two of these cursors by iteratively scanning the list until we find the node before the cursor. We visit the element at the cursor, and update the cursor to the previous node.
Each of these segments is $\sqrt{n}$ long, so each segment takes $\Theta(\sqrt{n^2})=\Theta(n)$ time to traverse. There are $\sqrt{n}$ many such segments, so the total time is $\Theta(n^{3/2})$.

Problem 2. Consider the problem of given an integer n, generating all possible permutations of the numbers {1, 2, …, n}. Provide a recursive algorithm for this problem

The helper function outputs permutations of the input array A that have the first i elements fixed.

```
def permutations-recursive(n):
     #input: integer n
     #do: print all permutations of order n
     def helper(A, i):
          if i = n then
               print(A)
          for j in [i to n] do
```

```
                swap A[i] and A[j]
                helper(A, i+1)
                swap A[i] and A[j]
A ← array([1, 2, …, n])
helper(A, 0)
```

The correctness of the algorithm hinges on the fact that while the helper function and its many recursive calls may modify the array during their execution, when a call to a helper function finally returns, the input array is always restored to the state it was in when the call started executing.
For a formal argument we prove by induction that the property that when we call helper(A, i) the algorithm outputs all of the array A that leaves the entries A[0 : i] fixed while trying all permutations of A[i : n].

Problem 3. Consider the problem of given an integer n, generating all possible permutations of the numbers {1, 2, …, n}. Provide a non-recursive algorithm for this problem using a stack.

Problem 4. Using only two stacks, provide an implementation of a queue. Analyse the time complexity of enqueue and dequeue operations.

The simplest solution is to push elements as they arrive at the first stack. When we are required to carry out a dequeue operation, we transfer all the elements to the second stack, pop once to later return the element on the top of the second stack (which is the 'last' element), and then transfer back all the remaining elements back to the first stack.

Correctness: This strategy works because when we transfer the elements from one stack to the next, we reverse the order of the elements. Before we transfer things, the most recent element to be queued is at the top of the first stack. After we transfer, the oldest element queue is at the top of the second stack. Finally, when we transfer the elements back to the first, we go back to the original stack order.

Time Complexity: If the queue holds n elements each enqueue operation takes O(1) time and each dequeue takes O(n) time since we need to transfer all n elements twice.

Problem 5. Design a queue with an operation GetAverage() that returns the average value of all elements stored in the queue. This operation should run in O(1) time and the running time of the other queue operations should remain the same as those of a regular queue.

(a) Describe your algorithm.

Recall that the average is the total sum of the elements divided by the number of elements. Since we already store the size of the queue, it suffices to add a single new variable that stores the sum of the elements, say S. When a new element gets enqueued or dequeued, the sum will be updated.

```
def getAverage():
     if isEmpty() then
          return "Queue empty"
     return sum / size
```

```
def newEnqueue(e):
      sum = sum + e
      enqueue(e)

def newDequeue():
      e <- dequeue
      sum = sum - e
      return e
```

(b) The correctness follows directly from the definition of average, assuming we maintain the sum correctly. Since we add the enqueued element's value to the sum and subtract it when it's dequeued, the sum is maintained correctly.

(c) The getAverage operation checks if the queue is empty, which takes O(1) time, and either throws an error (O(1) time) or returns the required division of two integers (O(1) time). Hence, the total running time is O(1) as required.
We modified the enqueue and dequeue operations. Adding the new element to the sum takes O(1) time, so enqueue still runs in O(1) time. Similarly, subtracting the removed element from the sum takes O(1) time, so dequeue still runs in O(1) time.

Trees

In computer science, a tree is an abstract model of a hierarchal structure, consisting of nodes with a parent-child relation.

Formal Definition
A tree T is made up of a set of nodes endowed with parent-child relationships with the following properties:

- If T is non-empty, it has a special node called the root that has no parent
- Every node v of T other than the root has a unique parent
- Following the parent relation always leads to the root

Terminology

1. **Ordered Tree**

A tree is ordered if there is a linear ordering defined for the children of each node; that is, we can identify children of a node as being the first, second, third, and so on. Ordered trees typically indicate the linear order relationship existing between siblings by listing them in the correct order. Examples: a structured document such as a book is hierarchically organised as a tree, whose internal nodes are chapters, sections, and whose external nodes are paragraphs, tables, bibliography etc.

2. **Binary Tree**

A binary tree is an ordered tree in which every node has at most two children.

Tree Traversal
Assumptions on the running times of various methods for a tree:

- The accessor methods root() and parent(v) take O(1) time
- The query methods isInternal(v), isExternal(v), and isRoot(v) take O(1) time as well

- The accessor method children(v) takes O(C_u) time where C_u is the number of children of v
- The generic methods swapElements(v, w) and replaceElement(v, e) take O(1) time
- The generic methods elements() and positions(), which return sets, take O(n) time, where n is the number of nodes in the tree

Depth and Height Example

Depth

Let v be the node of a tree T. The **depth** of v is the number of ancestors of v, excluding v itself. Note that this definition implies that the depth of the root of T is 0. The depth of a node v can also be recursively defined as follows:

- If v is the root, then the depth of v is 0
- Otherwise, the depth of v is one plus the depth of the parent of v.

Based on the above definition, the recursive algorithm below, computes the depth of a node v of T by calling itself recursively on the parent of v, and adding 1 to the value returned.

```
Algorithm depth(T, v):
     if T.isRoot(v) then
          return 0
     else
          return 1 + depth(T, T.parent(v))
```

The running time of algorithm depth(T, v) is O($1+d_v$) where d_v denotes the depth of node v in the tree T, because the algorithm performs a constant-time recursive step for each ancestor of v. Thus, in the worst case, the depth algorithm runs in O(n) time, where n is the total number of nodes in the tree T, since some nodes may have nearly this depth in T. Although such running time is a function of the input size, it is more accurate to characterise the running time in terms of the parameter d_v , since this will often be much smaller than n.

Height

The **height** of a tree T is the maximum depth of an external node of T. While this definition is correct, it does not lead to an efficient algorithm. Indeed, if we were to apply the above depth-finding algorithm to each node in the tree T, we would derive an O(n^2)-time algorithm to compute the height of T. We can be more efficient by using the following recursive definition of the height of a node v in a tree T:

- If v is an external node, then the height of v is 0
- Otherwise, the height of v is one plus the maximum height of a child of v

The algorithm below is expressed by a recursive method height(T, v) that computes the height of the subtree of T rooted at a node v.

```
Algorithm height(T, v):
     if T.isExternal(v) then
          return 0
     else
          h = 0
          for each child of T.children:
               h = max(h, height(T, w))
```

```
        return 1 + h
```

The height algorithm is recursive, and if it is initially called on the root of T, it will eventually be called once on each node of T. Thus, we can determine the running time of this method by an amortisation argument where we first determine the amount of time spent at each node (on the nonrecursive part), and then sum this time bound over all the nodes.
The computation of a set returned by children(v) takes O(C_u) time, where C_u denotes the number of children of node v. Also, the for loop has C_u iterations, and each iteration of the loop takes O(1) time plus the time for the recursive call on a child of v. Thus, the algorithm height spend O(1 + C_u) time at each node v and its running time is O($\sum_{v \in T} (1+C_u)$ ¿¿ .

To complete the analysis, we make use of the following property:
Let T be a tree with n nodes, and let C_u denote the number of children of a node v of T. Then,

$$\sum_{v \in T} C_u = n-1$$

Thus, by the theory above, the running time of algorithm height when called on the root of T is O(n).

Pre-Order Traversal
The preorder traversal algorithm is useful for producing a linear ordering of the nodes of a tree where the parents must always come before their children in the ordering.

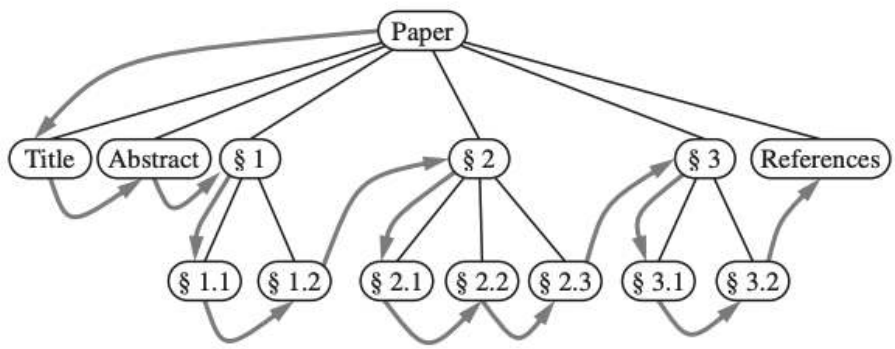


Figure 2.22: Preorder traversal of an ordered tree.

Example 2.6: *The preorder traversal of the tree associated with a document, as in Example 2.3, examines an entire document sequentially, from beginning to end. If the external nodes are removed before the traversal, then the traversal examines the table of contents of the document. (See Figure 2.22.)*

The overall running time of the preorder traversal of T is O(n).

Post-Order Traversal

Opposite of the preorder traversal, the postorder traversal recursively traverses the subtrees rooted at the children of the root first, and then visits the root. This method will visit a node v after it has visited all the other nodes in the subtree rooted at v.

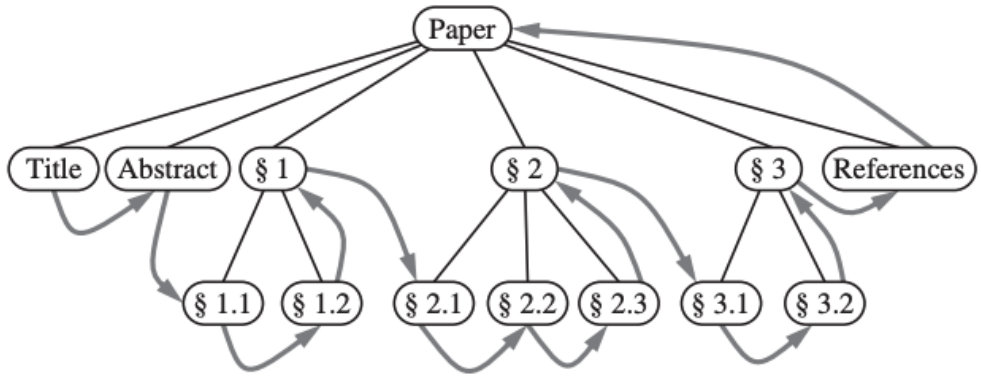


Figure 2.24: Postorder traversal of the ordered tree of Figure 2.22.

A postorder traversal of a tree T with n nodes takes O(n) time, assuming that visiting each node takes O(1) time. That is, the postorder traversal runs in linear time.

Binary Trees

Let T be a proper binary tree with n nodes, and let h denote the height of T. Then T has the following properties:

1. The number of external nodes in T is at least h + 1 and at most 2^h
2. The number of internal nodes in T is at least h and at most 2^h - 1
3. The total number of nodes in T is at least 2^h+1 and at most $2^{h+1}-1$
4. The height of T is at least log(n + 1) – 1 and at most (n – 1)/2

Theorem: In a property binary tree T, the number of external nodes is 1 more than the number of internal nodes.

The proof is by induction. If T itself only has one node v, then v is external, and the proposition clearly holds. Otherwise, we remove from T an (arbitrary) external node w and its parent v, which is an internal node. If v has a parent u, then we reconnect u with the former sibling z of w. This operation, removeAboveExternal(w), removes one internal node and one external node, and it leaves the tree being a proper binary tree. Thus, by the inductive hypothesis, the number of external nodes in this tree is one more than the number of internal nodes. Since we removed one internal and one external node to reduce T to this smaller tree, this same property must hold for T.

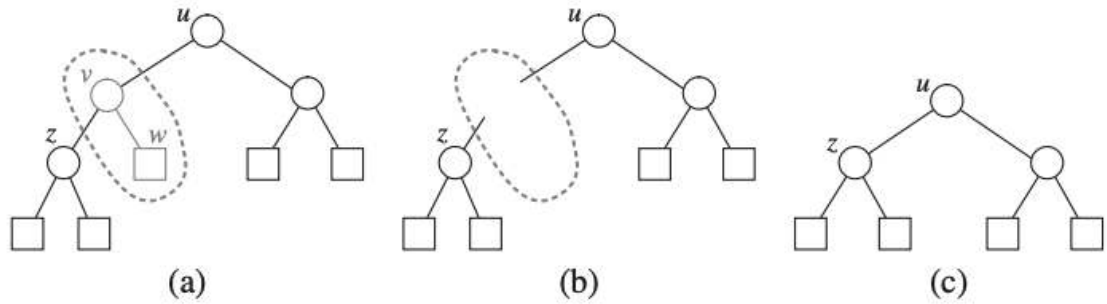


Figure 2.26: Operation removeAboveExternal(w), which removes an external node and its parent node, used in the justification of Theorem 2.8.

Note that the above relationship does not hold, in general, for nonbinary trees.

```
Algorithm binaryPreorder(T, v):
     perform the 'visit' action for node v
     if v is an internal node then
          binaryPreorder(T, T.leftChild(v))
          binaryPreorder(T, T.rightChild(v))

Algorithm binaryPostorder(T, v):
     if v is an internal node then
          binaryPostorder(T, T.leftChild(v))
          binaryPostorder(T, T.rightChild(v))
     perform the 'visit' action for the node v

Algorithm inOrder(T, v):
     if v is an internal node then
          inorder(T, T.leftChild(v)) //recursively traverse
left subtree
     perform the 'visit' action for node v
     if v is an internal node then
          inorder(T, T.rightchild(v)) //recursively traverse
right subtree
```

The addition inorder traversal of a binary tree T can be informally viewed as visiting the nodes of T "from left to right". Indeed, for every node v, the inorder traversal visits v after all the nodes in the left subtree of v and before all the nodes in the right subtree of v.

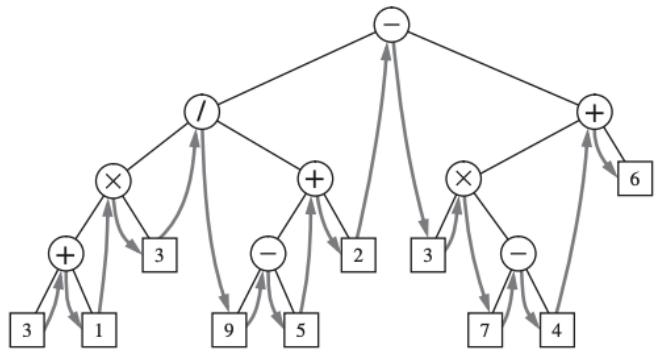


Figure 2.30: Inorder traversal of a binary tree.

Euler Tour Traversal of a Binary Tree

Can be informally defined as a 'walk' around T, where we start by going from the root towards its left child, viewing the edges of T as being 'walls' that we always keep to our left. Each node v of T is encountered three times by the Eulier tour: on the left, from below, and on the right.

If v is external, then these three "visits" actually all happen at the same time.

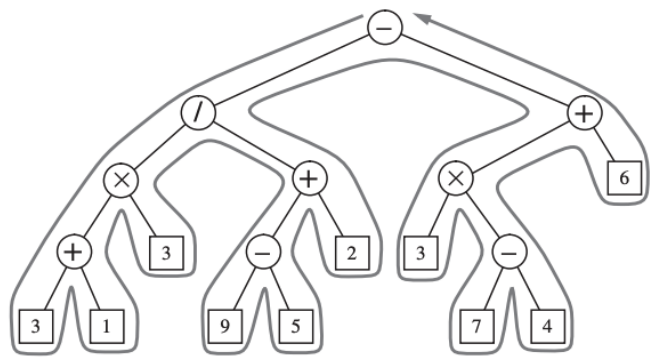


Figure 2.31: Euler tour traversal of a binary tree.

The Euler tour traversal extends the preorder, inorder and postorder traversals, but it can also perform other kinds of traversals. For example, if we wish to compute the number of descendants of each node v in an n node binary tree T.

We start an Euler tour by initialising a counter to 0, and then increment the counter each time we visit a node on the left. To determine the number of descendants of a node v, we compute the difference between the values of the counter when v is visited on the left and when it is visited on the right, and add 1. This simple rule gives us the number of descendants of v, because each node in the subtree rooted at v is counted between v's visit on the left and v's visit on the right. Therefore, we have an O(n) time method for computing the number of descendants of each node in T.

Assuming visiting a node takes O(1) time, the overall running time is O(n) for an n node tree.

Problems

Problem 1. Let A be an array holding n distinct integer values. We say that a tree T is a *pre-order realization of* A if T holds the values in A and a pre-order traversal of T visits the values

in the order they appear in A.
Design an algorithm that given an array produces a pre-order realization of it.

Problem 2. Design a linear time algorithm that given a tree T computes for every node u in T the size of the subtree rooted at u.

Problem 3. In a binary tree there is a natural ordering of the nodes on a given level of the tree (ie the left-to-right order) that you get when you draw the tree. Design an algorithm that given a tree T and a level k, visits the nodes in level k in this natural order. Your algorithm should perform the whole traversal in O(n) time.

Problem 4.

Problem 5.

Problem 6.

Binary Search Trees

The Problem: Supposed we are given an ordered set, S, of objects represented as key-value pairs and we would like to locate object x relative to the objects in this set. That is, we would like to identify the nearest neighbours of x in S: the object to the immediate left of x and immediate right of s, if these objects exist.

The Binary Search Algorithm (Sorted Array)
We call an item I of S a candidate if, at the current stage of the search, we cannot rule out that I has key equal to k. The following method maintains two parameters, low and high, such that all the candidate items have index at least low and at most high in S.
Initially, low = 1 and high = n, and we let key(i) denote the key at index i, which has elem(i) as its element. We then compare k to the key of the median candidate, that is, the item with index:

$$mid = \lfloor (low + high) \rfloor$$

We consider 3 cases:

1. If k = key(mid), then we have found the element we were looking for
2. If k < key(mid), then we recur on the first half of the vector, that is, on the range of indices from low to mid – 1
3. If k > key(mid), we recursively search the range of indices from mid + 1 to high

Binary Search in an Ordered Array:

- Input: an ordered array, A, storing n items, whose keys are accessed with method key(i) and whose elements are accessed with method elem(i); a search key k; and integers low and high

```
Algorithm BinarySearch(A, k, low, high):
     if low > high then
```

```
        return null
    else
        mid = (low + high) / 2
        if k = key(mid) then
            return elem(mid)
        else if k < key(mid) then
            return BinarySearch(A, k, low, mid-1)
        else
            return BinarySearch(A, k, mid+1, high)
```

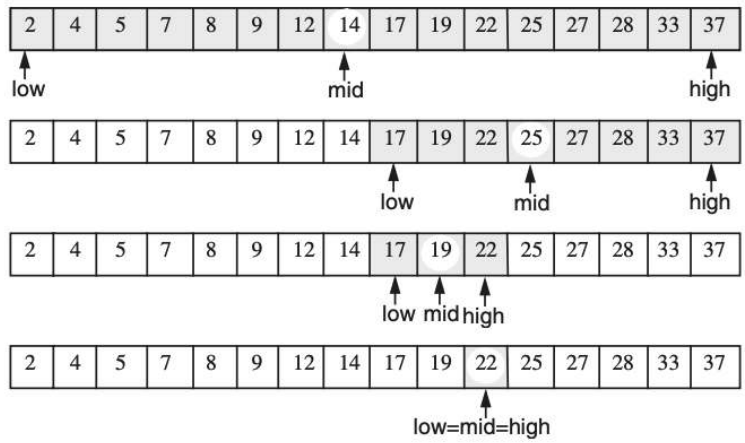


Figure 3.3: Example of a binary search to search for an element with key 22 in a sorted array. For simplicity, we show the keys but not the elements.

Analysing the Binary Search Algorithm

We observe a constant number of operations executed at each recursive call. Hence, the running time is proportional to the number of recursive calls performed. With each recursive call, the number of candidate items still to be searched in the array A is given by the value 'high – low + 1'. Moreover, the number of remaining candidates is reduced by at least one half with each recursive call.

Initially, the number of candidates is n; after the first call to BinarySearch, it is at most n/2; after the second call, it is at most n/4 and so on. If we let a function T(n) represent the running time of this method:

$$T(n) \leq \begin{cases} b & ; if\ n<2, else, \\ T(n/2)+b \end{cases}$$

Where b is a constant. In general, this recurrence equation shows that the number of candidate items remaining after each recursive call is at most $n/2^i$. In the worst case (unsuccessful search), the recursive calls stop when there are no more candidate items. Hence, the maximum number of recursive calls performed is the smallest integer m such that $n/2^m<1$. In other words, $m>\log n$. Thus, we have $m=\lfloor \log n \rfloor+1$, which implies the algorithm runs in O(logn) time.

The space requirement of this solution is Θ (n), which is optimal since we must store the n objects somewhere. This solution is only efficient if the set S is static, that is if we don't want to insert or delete any key-value pairs. In the dynamic case where we do need to perform insertions or deletions, then such updates take O(n) time. This is because of the need to move

elements in A greater than the insertion key in order to keep all the elements in A in sorted order.

Thus, using an ordered array to support fast searching only makes sense for the sake of efficiency if we don't need to perform insertions or deletions.

Binary Search Trees

We define a binary search tree to be a binary tree in which each internal node v stores an element e such that the elements stored in the left subtree of v are less than or equal to e, and the elements stored in the right subtree of v are greater than or equal to e. Assume external nodes store no elements; they could in fact be null.

An *inorder* traversal visits the elements stored in such a tree in *nondecreasing order*. Ie starting from the left-most child to the largest child on the right-most part of the tree.

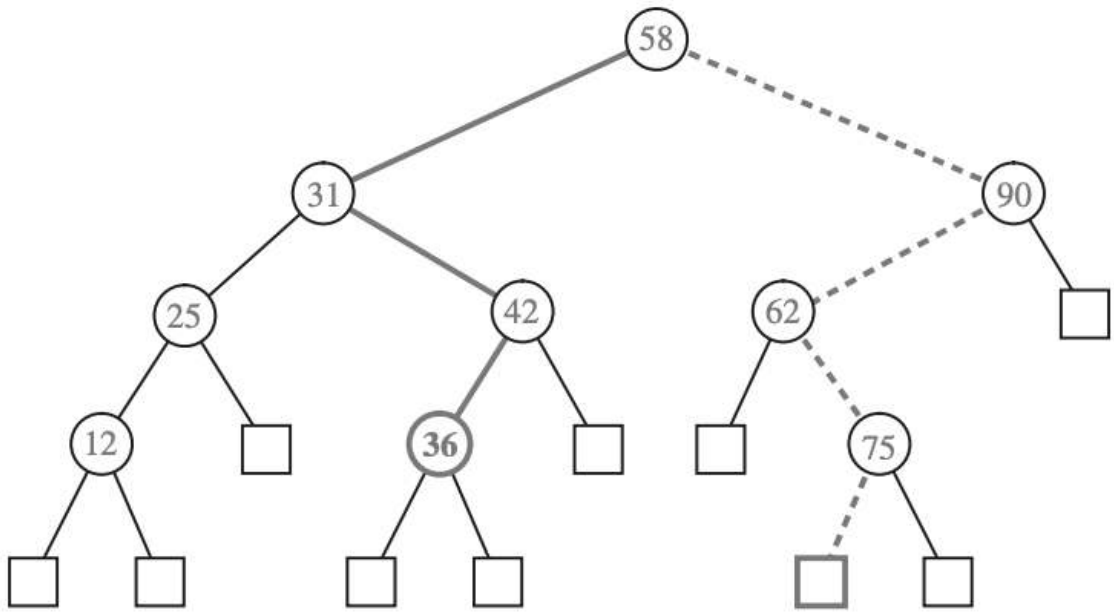


Figure 3.4: A binary search tree storing integers. The thick solid path drawn with thick lines is traversed when searching (successfully) for 36. The thick dashed path is traversed when searching (unsuccessfully) for 70.

```
Algorithm TreeSearch(k, v):
     if v is an external node then
          return v
     if k = key(v) then
          return v
     else if k < key(v) then
          return TreeSearch(k, T.leftChild(v))
     else
          return TreeSearch(k, T.rightChild(v))
```

The binary search tree executes a constant number of primitive operations for each node it traverses in the tree. Each new step in the traversal is made on the child of the previous node. That is, the binary search algorithm is performed on the nodes of a path T that starts from the

root and goes down one level at a time. Thus, the number of such nodes is bounded by height + 1. In other words, since we spend O(1) time per node encountered, the search method runs in O(h) time, where h is the height of the binary tree T.

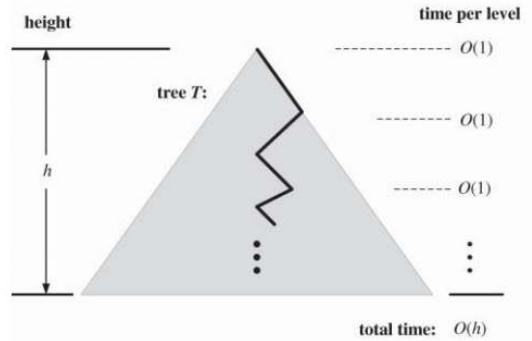


Figure 3.6: Illustrating the running time of searching in a binary search tree. The figure uses standard visualization shortcuts of viewing a binary search tree as a big triangle and a path from the root as a zig-zag line.

The height h of T can potentially be as large as n, but we expect that it is usually much smaller.

$$H(n) \leq \begin{cases} 1 & \text{if } n<2 \\ H(3n/4)+1 & \text{else.} \end{cases}$$

The child of the root stores at most (3/4)n items in its subtree, any of its children store at most $(3/4)^2 n$ items in their subtrees, any of their children store at most $(3/4)^3 n$ in their subtrees and so on. This implies that H(n) is at most $\lceil \log_{4/3} n \rceil$, which is O(logn). Intuitively, we achieve a logarithmic height for T because the subtrees rooted at each child of a node in T have roughly 'balanced' size.

Operations

1. Insertion

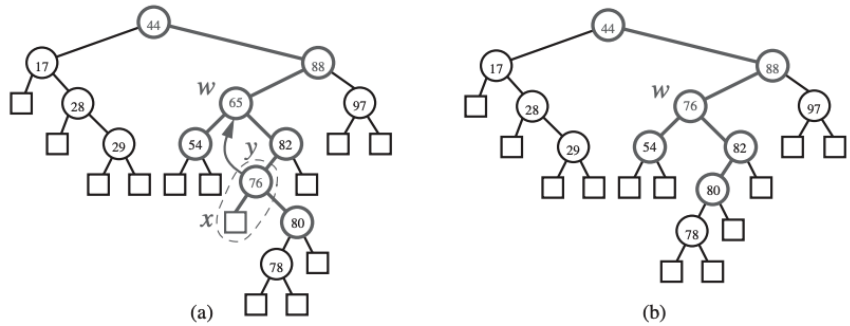
An insertion algorithm traces a path from the root of T down to an external node, w, which is the appropriate place to insert an item with key k based on the ordering of the items stored in T. This node then gets replaced with a new internal node accommodating the new item. Hence, an insertion adds the new item at the 'bottom' of the search tree T.

The analysis of the insertion algorithm mirror that for searching. The number of nodes visited is proportional to the height h of T in the worst case, since we spend O(1) time at each node visited. Thus, it runs in O(h) time.

2. Deletion

To avoid creating a 'hole' in a tree (in the case of deleting an internal node):

- We find the first external node y that follows w in an inorder traversal of T. Node y is the left post internal node of w, and node x if the left child of y.
- We save the element stored at w in a temporary variable t, and move the item of y into w. This removes the former item stored at w.
- We remove x and y from T by replacing y with x's sibling and removing both x and y from T (equivalent to operation removeAboveExternal(x) on T)



Running time is also O(h).

Performance of BST's

A BST is an efficient implementation of an ordered set of n key-value pairs, but only if the height of T is small. We would like the BST to have height O(logn), however, in some cases, it may be of linear height.

A binary search tree of heigh h storing n items supports range query operations with the following performance:

- The space used is O(n)
- The operation findAllInRange takes O(h + s) time, where s is the number of elements reported
- Operations insert and remove each take O(h) time

Problems

Problem 1. Consider the implementation of a binary search tree on n keys where the keys are stored in the internal nodes. Prove using induction that the height of the tree is at least $\log_2 n$.

Problem 2. Consider the implementation of a binary search tree on n keys where the keys are stored in the internal nodes. Prove using induction that the number of external nodes is n + 1.

Problem 3. Consider the following algorithm for testing if a given binary tree has the binary search tree property.

```
def test-best(T)
     for u in T do
          if u.left != null and u.key < u.left.key then
               return false
          if u.right != null and u.key > u.right.key then
               return false
     return true
```

Either prove the correctness of the algorithm or provide a counter example where it fails to return the correct answer.

Problem 4. Consider the following operation on a binary search tree: largest() that returns the largest key in the tree. Give an implementation that runs in O(h) time, where h is the height of the tree.

Problem 5. Consider the following operation on a binary search tree: Median() that returns the median element of the tree (one whose ranking is $\lfloor n/2 \rfloor$ in sorted order). Give an implementation that runs in O(h) time. You can augment the insertion and delete routines to keep additional data at each node.

Priority Queues

Hashing

Application: Network Routers

Network routers process multiple streams of packets at a high speed. To process a packet with destination k and data payload x, a router must determine which outgoing link to send the packet along.

Such a system needs to support:

- Destination-based lookups ie get(k) operations that return the outgoing link for destination k
- Updates to the routing table ie put(k, c) operations, where c is the new outgoing link for destination k

Ideally, we would like to achieve O(1) time for both operations.

The hash table data structure is able to achieve O(1) expected time for both get and put operations.

Maps

The main idea at the heart of the hash table data structure is that it allows users to assign keys to element and then use those keys later to look up or remove elements. This functionality defines a data structure known as a dictionary or map.

A map stores a collection of key-value pairs (k, v) ('items') where k is a key and v is a value associated with that key. In most applications, keys are unique. Maps that allow for multiple values associated to the same key are known as a multimap.

A map data structure supports the following fundamental methods:

- Get(k)
- Put(k, v)
- Remove(k)

Lookup Tables

This implementation views key-value pairs, under the assumption each key is unique, as an array. The key k allows us to simply 'look up' the item for k by a single array-indexing operation.

To perform a put(k, v) operation, we assign (k, v) to A[k]

To perform a get(k) operation, we return A[k]

To perform a remove(k) operation, we return A[k] and assign a NULL item to A[k]

The analysis of the performance of a lookup table is also simple. Each of the essential map methods run in O(1) time in worst case. Thus, in terms of time performance, the lookup table is optimal.

However, in terms of space usage, we note that the total amount of memory used for a lookup table is Θ (N). We refer to the size N of the array A as being the capacity of this map implementation. Such an amount of space is reasonable if the number of items, n, is close to the capacity, N.
Another drawback of this method is that it requires keys to be unique integers in the range [0, N-1], which is often not the case. We need a mechanism to overcome these drawbacks while still achieving simple and fast methods for implementing the essential operations for a map.

Hash Functions
If we cannot assume that keys are unique integers in the range [0, N-1], then we need a good way of assigning keys to integers in this range. The hash function, h, maps key k in our map to an integer in the range [0, N-1], where N is the capacity of the underlying array for this table. The use of such a function allows us to treat objects, such as strings, as numbers.

Summing Components

Modular Division

Universal Hash Functions

Random Linear Hash Function

Collision Handling

Separate Chaining

Open Addressing using Linear Probing

Search with Linear Probing

Cuckoo Hashing

Problems

Graphs

Edge Types

Undirected Graphs

Directed Graphs

Graph Abstract Data Type
We model the abstraction as a linear combination of three data types: vertex, edge, and graph
A vertex stores an associated object that is retrieved with a getElement() method.
An edge stores an associated object that is retrieved with a getElement() method

Problems

Problem 1. An undirected graph G = (V, E) is said to be bipartite if its vertex set V can be partitioned into two sets A and B such that E $\subseteq A \times B$. Design an O(n + m) algorithm to test if a given input graph is bipartite using the following guide:

a) Suppose we run BFS from some vertex $s \in V$ and obtain layers $L_1, \ldots, L_k$. Let (u,v)be some edge in E. Show that if $u \in L_i$ and $v \in L_j$ then $|i-j| \leq 1$.

If i = j then we are done. Otherwise, assume without loss of generality that u is discovered by BFS before v; in other words, assume i < j. When processing u, BFS scans the neighbourhood of u. Since v ends up in layer Lj and j > i, this means that v had not been discovered yet at the time we started processing u. But then that means that v will be placed in the next layer since (u, v) is an edge; in other words, j = i + 1, and the property follows

b) Suppose we run BFS on G. Show that if there is an edge (u, v) such that u and v belong to the same layer then the graph is not bipartite.

c) Suppose G is connected and we run BFS. Show that if there are no intra-layer edges then the graph is bipartite.

d) Put together all the above to design an $O(n + m)$ time algorithm for testing bipartiness.

Problem 2.

Problem 3.

Problem 4.

Graph Algorithms

Dijkstra's Algorithm

```
Def Digjkstra(G, w, s):
     #initialise algorithm
     for v in V:
          D[v] <- infinity
          Parent[v] <- empty set
     D[s] <- empty set
     Q <- new priority queue for {(v, D[v]) : v in V}

     #iteratively add vertices to S
     While Q is not empty:
          u <- Q.remove_min()
```

```
            for z in G.neighbours(u):
                  if D[u] + w[u, z] < D[z]:
                        D[z] <- D[u] + w[u, z]
                        Q.update_priority(z, D[z])
                        Parent[z] <- u
      Return D, parent
```

Assuming the graph is connected (so m >= n – 1), the algorithm spends O(m) time on everything except PQ operations.
Priority queue operation counts:

- Insert: n
- Decrease_key: m
- Remove_min: n

Using a heap for priority queue, Dijkstra runs in O(mlogn) time.
Fibonacci heap is a PQ that can carry out decrease key in O(1) amortized time. Using that instead we get O(m + nlogn) time.

Minimum Spanning Tree

Assumption: all edge costs are distinct
Cut property: Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST must contain e
Cycle property: Let C be any cycle and let f be the max cost edge belonging to C. Then the MST does not contain f.

Prim's Algorithm

```
Def prim(G, c):
      U <- arbitrary vertex in V
      S <- { u }
      T <- empty set
      while |S| <|V| do
            (u, v) <- min cost edge such that u in S and v not in S
            Add (u, v) to T
            Add v to s
      Return T
```

Every time we add an edge we follow the cut property. The main idea is that for every vertex v in V \ S, we keep:

- D[v] = distance to closest neighbour in S
- Parent[v] = closest neighbour in S

Walk through idea:

- First we initialise array, with S = u, distance = infinity, parent = null
- Starting with any node, we update the distances of adjacent, unselected nodes
- We select the node with the minimum distance
- Repeat the process to attain the cost of the MST

Time complexity: O(mlogn) using a heap. O(m + nlogn) using Fibonacci heap.

Kruskal's Algorithm

Consider edges in ascending order of weight.
Case 1: If adding e to T creates a cycle, discard e according to the cycle property.

Case 2: Otherwise, insert e = (u, v) into T according to cut property where S = set of nodes in u's connected component.
Sorting edges takes O(mlogm) time. We need to be able to test if adding a new edge creates a cycle, in which case we skip the edge. One option is to run DFS in each iteration to see if the number of connected components stays the say. This leads to O(mn) time for the main loop.

Union Find ADT
Data structure defined on a ground set of elements A. Used to keep track of an evolving partition of A. Operations include:

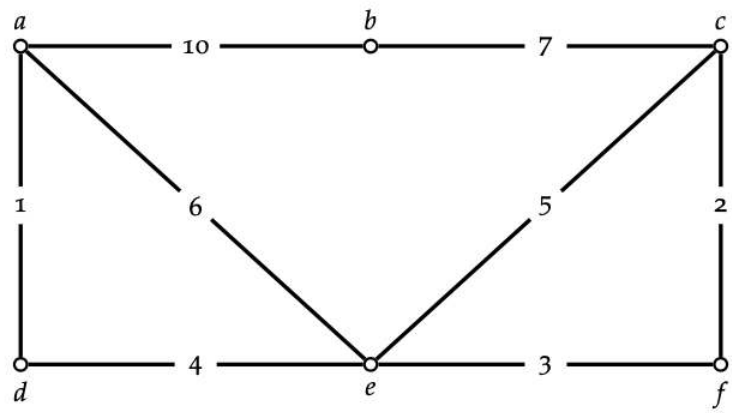
- Make_sets(A): make |A| singleton sets with elements in A → O(n) time
- Find(A): returns an id for the set element a belongs to → O(1) time
- Union(a, b): union the sets elements a and b belong to → O(n) time

Kruskal's algorithm would run in O(n2) time after the edge weights are sorted

Problem 1. Consider Dijkstra's shortest path algorithm for undirected graphs. What changes (if any) do we need to make to this algorithm for it to work for directed graphs and maintain its running time?

The only thing we need to consider is that when updating D[z], we need to consider only the vertices where there is a directed edge from u to z. In the pseudocode in the lecture, this is implicitly handles by only considering the neighbours of u (ie the vertices that can be reached by following an edge from u) so we don't need to change anything

Problem 2. Consider the following weighted undirected graph *G*:



Your task is to compute a minimum weight spanning tree *T* of *G*:

a) Which edges are part of the MST?
(a, d), (d, e), (e, f), (c, f), (b, c)

b) In which order does Kruskal's algorithm add these edges to the solution.
The edges are considered in order of their weight, so: (a, d), (c, f), (e, f), (d, e), (b, c)

Problem3. Let $G=(V,E)$ be an undirected graph with edge weights $w : E \rightarrow R_+$. For all $e \in E$, define $w_1(e) = \alpha w(e)$ for some $\alpha > 0$, $w_2(e) = w(e)+\beta$ for some $\beta>0$, and $w_3(e)=w(e)^2$.

a) Suppose p is a shortest s-t path for the weights w. Is p still optimal under w_1? What about under w_2? What about under w_3?

b) Suppose T is minimum weight spanning tree for the weights w. Is T still optimal under w_1? What about under w_2? What about under w_3?

Problem 4. It is not uncommon for a given optimization problem to have multiple optimal solutions. For example, in an instance of the shortest s-t path problem, there could be multiple shortest paths connecting s and t. In such situations, it may be desirable to break ties in favor of a path that uses the fewest edges.

Show how to reduce this problem to a standard shortest path problem. You can assume that the edge lengths l are positive integers.

a) Let us define a new edge function $l'(e) = Ml(e)$ for each edge e. Show that if P and Q are two s-t paths such that $l(P) < l(Q)$ then $l'(Q) - l'(P) \geq M$.

b) Let us define a new edge function $l''(e) = Ml(e) + 1$ for each edge e. Show that if P and Q are two s-t paths such that $l(P) = l(Q)$ but P uses fewer edges than Q then $l''(P) < l''(Q)$.

c) Show how to set M in the second function so that the shortest s-t path under l'' is also shortest under l and uses the fewest edges among all such shortest paths.

Problem 5. Consider the following generalization of the shortest path problem where in addition to edge lengths, each vertex has a cost. The objective is to find an s-t path that minimizes the total length of the edges in the path plus the cost of the vertices in the path. Design an efficient algorithm for this problem.

Greedy Algorithms

The greedy algorithm design repeated makes choices that optimise some objective function, like repeatedly accepting the bid that maximises the price-per-unit. The trick in applying this technique is guaranteeing that such a local 'greedy' choice will always lead to the optional solution.
Proving that the greedy method can indeed lead to an optimal solution can often require deeper understanding of the problem being solved, so we may need to make additional assumptions for it to work.

The Greedy Method

The greedy method is applied to optimisation problems – that is, problems that involve searching through a set of configurations to find one that minimises or maximises an objective function defined in these configurations.

The general formula is as follows: in order to solve a given optimisation problem, we proceed by a sequence of choices. The sequence of choices starts from some well-understood starting configuration, and then iteratively makes the decision that is best from all of those that are currently possible, in terms of improving the objective function

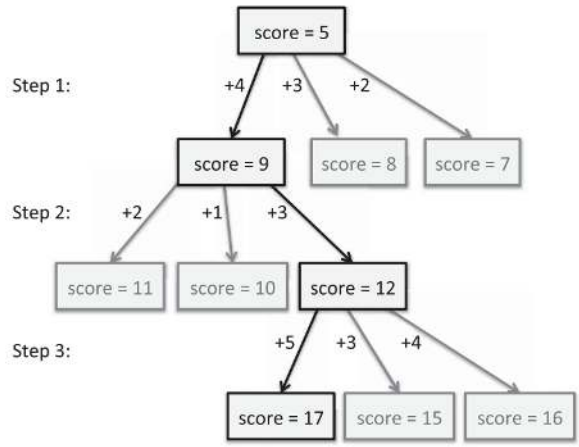


Figure 10.1: An example of the greedy method. Each rectangle represents a configuration whose score is the objective function. A configuration has three choices, each of which provides a different incremental improvement to the score. The bold choices are the ones that are picked in each step according to the greedy strategy.

The greedy approach works optimally for problems which possess the **greedy choice** property. This is a property that a global optimal configuration can be reached by a series of locally optimal choices (that is, choices that are the best from among the possibilities available at the time), starting from a well defined configuration.

The Fraction Knapsack Problem

We are given a set S of n items, such that each item i has a positive benefit b_i and a positive weight w_i , and we wish to find the maximum benefit subset that does not exceed a given weight W. If we are restricted to entirely accepting or rejecting each item, then we would have the 0-1 version of this problem. Let us now allow ourselves to take arbitrary fractions of some elements, however.

The motivation for this knapsack problem is that we are going on a trip and we have a single knapsack that can carry items of a total weight at most W. In addition, we are allowed to break items into fractions arbitrarily. That is, we can take an amount x_i of each item i such that

$$0 \le x_i \le w_i \, for \, each \, i \in S \wedge \sum_{i \in S} x_i \le W$$

The total benefit of the items taken is determined by the objective function

$$\sum_{i \in S} b_i (x_i / w_i)$$

Consider, for example, a student who is going to an outdoor sporting event and must fill a knapsack full of foodstuffs to take along. As long as each candidate foodstuff is something that can be easily divided into fractions, such as drinks, potato chips, pizza, then this would be an instance of the fractional knapsack problem.

In applying the greedy method, the most important decision is to determine the objective function that we wish to optimise. For example, we could choose to include items into our knapsack based on their weights, say, taking items in order by increasing weights. The

intuition behind this approach is that the lower-weight items consume the least amount of the weight resource of the knapsack. Unfortunately, this first idea doesn't work.
The best approach is to rank the items by their value, which we define to be the ratio of their benefits and weights. The intuition behind this is that benefit-per-weight-unit is a natural measurement of the inherent value that each item possess. Indeed this approach leads to an efficient algorithm that finds an optimal solution to the knapsack problem.

```
Algorithm FractionalKnapsack(S, W):
     for each item i in S:
          x(i) = 0
          v(i) = b(i)/w(i) //value index of item i
     w = 0 //total weight
     while w < W and S is not empty:
          remove from S an item i with highest value index
          a = min {w(i), W - w}
          x(i) = a
          w = w + a
```

The FractionalKnapsack algorithm can be implemented O(nlogn) time, where n is the number of items in S. Specifically, we can use a heap-based priority queue to store the items of S, where the key of each item is its value index. With this data structure, each greedy choice, which removes the item with the greatest index value, takes O(logn) time.

Task Scheduling
Supposed we are given a set T of n tasks such that each task i has a start time, s_i , and a finish time, $f_i (where\, s_i < f_i)$. Task i must start at time s_i and must finish by time f_i . Each task has to be performed on a machine and each machine can execute only one task at a time. Two tasks i and j are said to be nonconflicting if they do not overlap in time. Clearly, two tasks can be scheduled to be executes on the same machine only if they are nonconflicting.

The task scheduling problem we consider here is to schedule all the tasks in T on the fewest machines possible in a non-conflicting way. Alternatively, we can think of the tasks as meetings that we must schedule in as few conference rooms as possible.

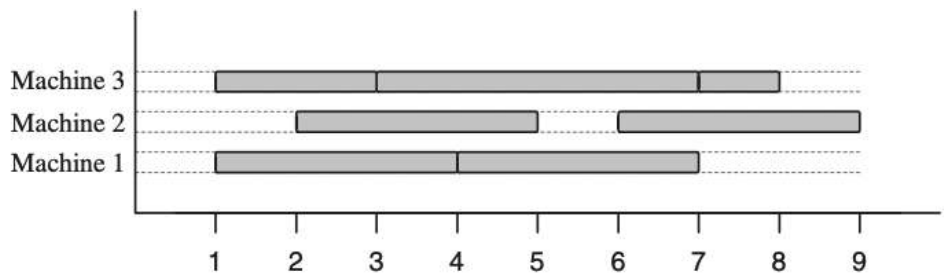


Figure 10.4: An illustration of a solution to the task scheduling problem, for tasks whose collection of pairs of start times and finish times is $\{(1,3), (1,4), (2,5), (3,7), (4,7), (6,9), (7,8)\}$.

As with any greedy strategy, the challenge is to find the right objective function. For example, we might consider the tasks from longest to shortest, assignment them to machines based on the first one available, since the longest tasks seem like they would be the hardest to schedule. Unfortunately, this approach does not necessarily result in an optimal solution.

The optimal greedy approach is to consider the tasks order by increasing start times. In this case, we would consider each task by the order of its start time, and assign it to the first machine that is available at that time. If there are no available machines, however, then we would need to allocate a new machine and schedule this task on that machine. The intuition behind this approach is that, by processing tasks by their start times, when we process a task for a given start time we will have already processed all the other tasks that would conflict with this starting time.

```
Algorithm TaskSchedule(T):
M = 0 //optimal number of machines
while T is not empty:
      remove from T task i with the smallest start time s(i)
      if there is a machine j with no task conflicting with task i
then:
            schedule task i on machine j
      else:
            m = m + 1 //add new machine
            schedule task i on machine m
```

The above greedy algorithm starts with no machines, then considers the tasks in a greedy fashion, ordered by their start times. For each task i, if we have a machine that can handle task i, then we schedule the task on that machine, say, choosing the first such available machine. Otherwise, we allocation a new machine, schedule i on it, and repeat this greedy selection process until we have considered all the tasks in T.

The following theorem states that greedy method TaskSchedule produces an optimal solution, because we are always considering a task starting at a given time after we have already processed all the tasks that might conflict with this start time.

Given a set of n task specified by their start and finish times, the algorithm TaskSchedule produces, in O(nlogn) time, a schedule for the tasks using a minimum number of machines.

Proof: Let k be the last machine allocated by algorithm TaskSchedule, and let i be the first task scheduled on k. When we scheduled I, each of the machines 1 through k – 1 contained tasks that conflict with i. S

Text Compression: Huffman Encoding

Given a string X, the goal is to efficiently encode X into a smaller string Y (saves memory and/or bandwidth).

This method produces a variable-length prefix code for X based on the construction of a proper binary tree T that represents the code. Each edge in T represents a bit in code word, with each edge to a left child representing a “0” and each edge to a right child representing a

"1". Each external node v is associated with a specific character, and the code word for that character is defined by the sequence of bits associated with the edges in the path from the root of T to v.

(a)

Character		a	b	d	e	f	h	i	k	n	o	r	s	t	u	v
Frequency	9	5	1	3	7	3	1	1	1	4	1	5	1	2	1	1

(b)

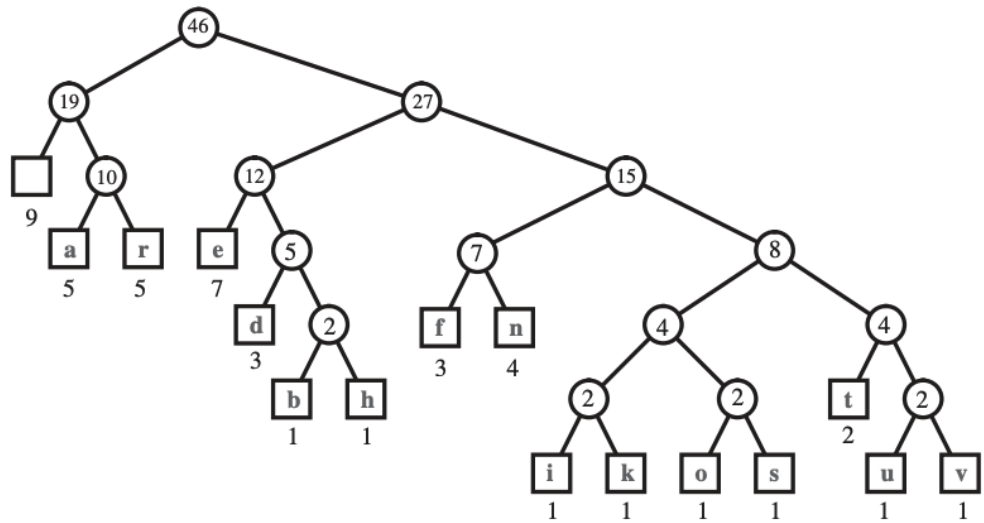


Figure 10.8: An illustration of an example Huffman code for the input string $X =$ `"a fast runner need never be afraid of the dark"`: (a) frequency of each character of X; (b) Huffman tree T for string X. The code for a character c is obtained by tracing the path from the root of T to the external node where c is stored, and associating a left child with 0 and a right child with 1. For example, the code for "a" is 010, and the code for "f" is 1100.

Problems

Problem 1. During the lecture we saw that the Fractional Knapsack algorithm's correctness depends on which greedy choice we make. We saw that there are instances where always picking the "highest benefit" or always picking the "smallest weight" doesn't give the optimal solution.
For each of these two strategies, give an infinite class of instances where the algorithm gives the optimal solution, thus showing that *you can't show correctness of an algorithm by using a finite number of example instances*.

The trivial solution that works for both strategies is any instance consisting of a single item with weight at most W (or any instance where the sum of the weights of all items is at most W).

A less trivial solution where not all items fit in the knapsack could for example be the following. For the “highest benefit” strategy, we construct the following class of instances:

- We have n items (numbered 0 to n – 1) and item i has benefit $b_i = c \times 2^i$ (for all c > 0) and weight $w_i = 2^i$.
- For any integer W, picking the highest benefit items that fit in the knapsack corresponds to picking those i that are 1 in the binary representation of W, filling the knapsack entirely. The solution will have benefit c x W.

Problem 2. Given vectors $a, b \in R^n$, consider the problem of finding a permutation a' and b' of a and b respectively that minimizes

$$\sum_{1 \le i \le n} \text{¿} ai - bi \vee .$$

Prove or disprove the correctness (i.e., that it always returns the optimal solution) of the following algorithm that greedily tries to pair up similar coordinates.

The algorithm does not always return the optimal solution. Consider the instance A = (1, 4) and B = (3, 6). Since the difference between 3 and 4 is the pair that minimises their difference, the algorithm ends up with the following pairs: (3, 4) and (1, 6). These pairs have the length of 6, whereas the optimal solution would have a difference of 4 using (1, 3) and (4, 6).

Problem 3. Given vectors a, b in R, consider the problem of finding a permutation a’ and b’ of a and b respectively that minimises

$$\sum_{1 \le i \le n} \text{¿} ai - bi \vee .$$

```
Def greedy-matching-v1(a, b):
     A <- multiset of values in a
     B <- multiset of values in b
     a’ <- new empty list
     b’ <- new empty list
     while A is not empty:
          x, y <- a pair in (A, B) minimising |x - y|
          append x to a’
          append y to b’
          remove x from A and y from B
     return (a’, b’)
```

The above algorithm sorts the coordinates of a and b in non-decreasing order. In this case, the algorithm is optimal.

Let a* and b* be the optimal solution. We use an exchange argument. First note that we can assume that a* is listed in non-decreasing order, because applying the same permutation to both a* and b* does not change the value of the objective. If b* is not in sorted order, there must be an inversion, that is, an index i such that $b*_{i+1}$ $b*_i > $. The contribution of the indices i and i + 1 to the objective is

$|a*_i - b*_i| + |a*_{i+1} - b*_{i+1}|$

Since we know a* is listed in non-decreasing order (smallest to largest), that is, $a*_i \le a*_{i+1}$, a case analysis argument can be used to show that

$|a*_i - b*_i| + |a*_{i+1} - b*_{i+1}| \ge |a*_i - b*_{i+1}| + |a*_{i+1} - b*_i|$

Therefore, by swapping entries $b*_i$ and $b*_{i+1}$, we have one fewer inversion without increasing the objective value.

Problem 4. Suppose we are to construct a Huffman tree for a string over an alphabet C = $c_1 + c_2 + \ldots + c_k$ with frequencies of $f_i = 2^i$. Prove that every internal node in T has an external-node child.

We prove by induction that at the start of the ith iteration of the algorithm, the set of trees the algorithm has is { $c_1, \ldots, c_i$ } , { c_{i+1} }, … { c_k }. The base case i = 1 is trivial, since all trees are singletons. For the induction step, assume

Problem 5. Supposed we are to schedule print jobs on a printer. Each job j has an associated weight w_j > 0 (representing how important the job is) and a processing time t_j (representing how long the job takes). Let C_j^{σ} be the completion time of job j under the schedule σ .
Design a greedy algorithm that computes a schedule minimising the sum of weighted completion times.

An optimal schedule can be obtained by sorting the jobs in increasing time/weight order. Assume for simplicity that no two jobs have the same ratio.

To show that the schedule is optimal, we use an exchange argument. Suppose the optimal solution is σ and there are two consecutive jobs, say i and k such that t/w(i) > t/w(k). We build another schedule T where we swap the positions of i and k, and leave the other jobs unchanged. We need to argues that this change decreases the cost of the schedule. Notice that the competition time of jobs other than i and k is the same in σ and T.

Divide and Conquer

Divide and conquer algorithms can normally be broken into three parts:

1. Divide: if it is a base case, solve directly, otherwise break up the problem into several parts
2. Recursively solve each sub problem
3. Conquer: combine the solutions of each part into the overall solution

We can model the divide-and-conquer approach by using a parameter n to denote the size of the original problem, and let S(n) denote this problem. We solve the problem S(n) by solving a collection of k subproblems S(n_1)… S(n_k) and then merging the solutions to these subproblems

Recurrence Formula

To analyse the running time of a divide and conquer algorithm, we often use a recurrence equation. That is, we let a function T(n) denote the running time of the algorithm on an input of size n, and we characterise T(n) using an equation that relates T(n) to values of the function T for problem sizes smaller than n.

Iterative Substitution Method

One way to solve a recurrence equation is to use the iterative substitution method. We assume that the problem size n is fairly large and we substitute the general form of the recurrence for each occurrence of the function T on the right-hand side. For example, using the merge-sort recurrence equation

$$T(n)=\begin{cases} b & \text{if } n<2 \\ 2T(n/2)+bn & \text{if } n\geq 2 \end{cases}$$

This yields the following equation

$$T(n)=2(2T(n/2^2)+b(n/2))+bn$$
$$¿\ 2^2T(n/2^2)+2bn$$

Plugging the general equation for T in again yields the equation

$$T(n)=2^2(2T(n/2^3)+b(n/2^2))+2bn$$
$$¿\ 2^3T(n/2^3)+3bn$$

The goal in applying the iterative substitution method is that at some point we will see a pattern that can be converted into a general closed-form equation (with T only appearing on the left-hand side).

$$T(n)=2^iT(n/2^i)+ibn$$

Note that the general form of this equation shifts to the base case, T(n) = b, when n = 2^i , that is, when i = logn, which implies

$$T(n)=bn+bnlogn$$

In other worse, T(n) is O(nlogn)

The Recursion Tree

Another way of characterising recurrence equations is to use the recursion tree method. This technique uses repeated substitution to solve a recurrence equation in a visual approach.

Example 11.1: *Consider the following recurrence equation:*

$$T(n) = \begin{cases} b & \text{if } n < 3 \\ 3T(n/3) + bn & \text{if } n \geq 3. \end{cases}$$

This is the recurrence equation that we get, for example, by modifying the merge-sort algorithm so that we divide an unsorted sequence into three equal-sized sequences, recursively sort each one, and then do a three-way merge of three sorted sequences to produce a sorted version of the original sequence. In the recursion tree R for this recurrence, each internal node v has three children and has a size and an overhead associated with it, which corresponds to the time needed to merge the subproblem solutions produced by v's children. We illustrate the tree R in Figure 11.2. Note that the overheads of the nodes of each level sum to bn. Thus, observing that the depth of R is $\log_3 n$, we have that $T(n)$ is $O(n \log n)$.

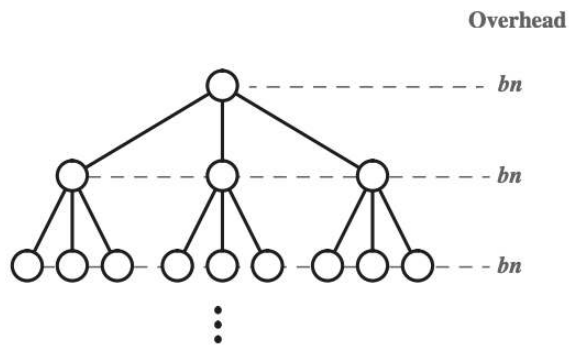


Figure 11.2: The recursion tree R used in Example 11.1, where we show the cumulative overhead of each level.

Master Theorem

Some Example Applications of the Master Theorem

We illustrate the usage of the master theorem with a few examples (with each taking the assumption that $T(n) = c$ for $n < d$, for constants $c \geq 1$ and $d \geq 1$).

Example 11.5: *Consider the recurrence*

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

In this case, $n^{\log_b a} = n^{\log_2 4} = n^2$. *Thus, we are in Case 1, for* $f(n)$ *is* $O(n^{2-\epsilon})$ *for* $\epsilon = 1$. *This means that* $T(n)$ *is* $\Theta(n^2)$ *by the master theorem.*

Example 11.6: *Consider the recurrence*

$$T(n) = 2T(n/2) + n \log n,$$

which is one of the recurrences given above. In this case, $n^{\log_b a} = n^{\log_2 2} = n$. *Thus, we are in Case 2, with* $k = 1$, *for* $f(n)$ *is* $\Theta(n \log n)$. *This means that* $T(n)$ *is* $\Theta(n \log^2 n)$ *by the master theorem.*

Example 11.7: *Consider the recurrence*

$$T(n) = T(n/3) + n,$$

which is the recurrence for a geometrically decreasing summation that starts with n. *In this case,* $n^{\log_b a} = n^{\log_3 1} = n^0 = 1$. *Thus, we are in Case 3, for* $f(n)$ *is* $\Omega(n^{0+\epsilon})$, *for* $\epsilon = 1$, *and* $af(n/b) = n/3 = (1/3)f(n)$. *This means that* $T(n)$ *is* $\Theta(n)$ *by the master theorem.*

Proof by Unrolling

Merge-Sort

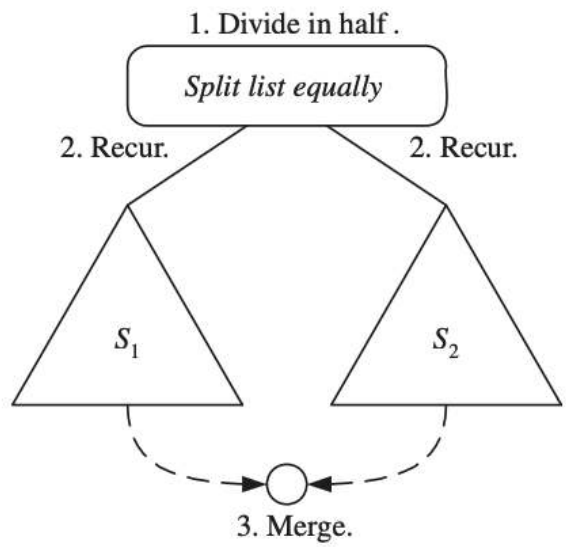


Figure 8.1: A visual schematic of the divide-and-conquer paradigm, applied to a problem involving a list that is divided equally in two in the divide step.

S(n) denotes the problem of sorting a sequence of n numbers. Merge-sort solves problem S(n) by dividing it into two subproblems S($\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$, recursively solving these two

subproblems, and then merging the resulting sorted sequences into a single sorted sequence that yields a solution to S(n).

```
Algorithm merge(s1, s2, s):
      //s1 and s2 sorted in non decreasing order, empty array s
      //output: S containing elements from s1 and s2 in sorted order
      i <- 1
      j <- 1
      while i <= n and j <= n:
            if s1[i] <= s2[j]:
                  s[i + j - 1] <- s1[i]
                  i++
            else:
                  S[i + j - 1] <- s2[j]
                  j++
      while i <= n:
            S[i+j - 1] <- s1[i]
            i++
      while j <= n:
            S[i+j - 1] <- s2[j]
            j++
```

Algorithm merge has three while loops. The key operations performed inside each loop takes O(1) time each. The key observation is that during each iteration of any one of the loops, one element is added to the output array S and is never considered again. This observation implies that the overall number of iterations of the three loops is n1 + n2. Thus, the running time of merge is O(n1 + n2).

Running Time

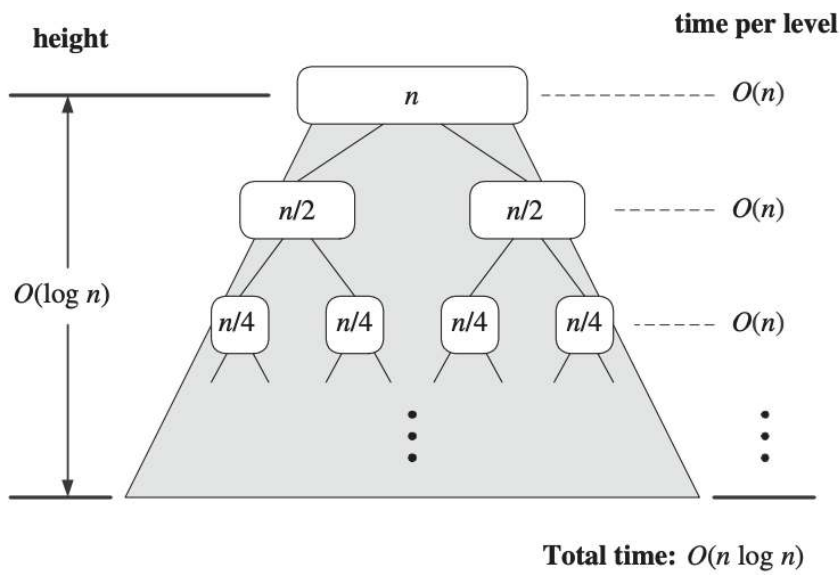


Figure 8.5: A visual analysis of the running time of merge-sort. Each node of the merge-sort tree is labeled with the size of its subproblem.

The merging step takes O(n) time. Thus, the total running time of the algorithm is O(nlogn). To analyse the running time formally, we use a recurrence equation:

$$T(n)=\begin{cases} b & \text{if } n<2 \\ 2T(n/2)+bn & \text{if } n\geq 2 \end{cases}$$

For some constant b, taking the simplifying assumption that n is a power of 2.

Quick Sort

Sorts a sequence S using a simple divide and conquer approach, whereby we divide S into sub sequences, recur to sort each subsequence, and then combine the sorted subsequences by a simple concatenation.

1. Divide: If S has at least two elements, selected a specific element x from S, which is called the **pivot**. A common practice is to choose the pivot to be the last element in S. Remove all the elements from S and put them into three sequences:
    - L, storing the elements in S less than x
    - E, storing the elements in S equal to x
    - G, storing the elements in S greater than x
2. Recur: Recursively sort sequences L and G
3. Conquer: Put the elements back into S in order by first inserting the elements of L, then those of E, and finally those of G

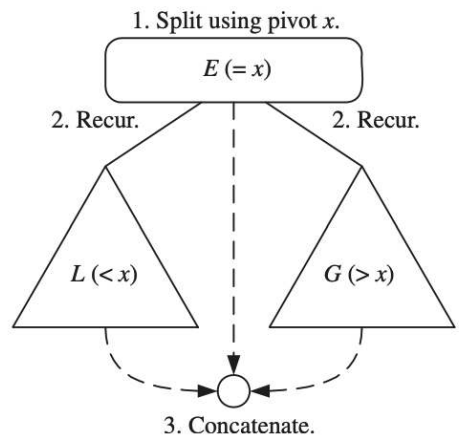


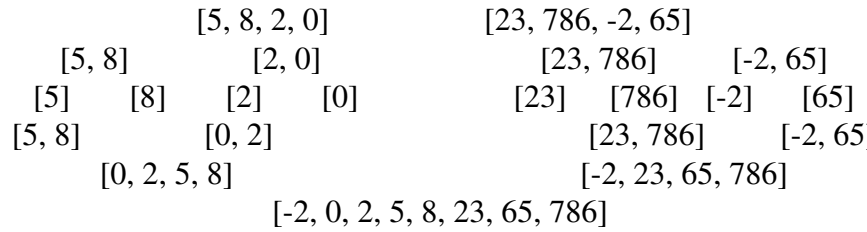
Figure 8.6: A visual schematic of the quick-sort algorithm.

The running time at each node v of binary tree T is proportional to the input size of v, defined as the size of the sequence handled by the invocation of quick sort associated with node v.

Theorem: The expected running time of randomised quick sort on a sequence of size n is O(nlogn).

Problems

Problem 1. Sort the following array using merge-sort: A = [5, 8, 2, 0, 23, 786, -2, 65]. Give all arrays on which recursive calls are made and show how they are merged back together. Until all arrays have size 1, all splits represent recursive calls. Since arrays of size 1 are sorted, after this the arrays are merged back together in order to create the final sorted array.



Problem 2. Consider the following algorithm.

```
Def reverse(A):
     if |A| = 1 then
          return A
     else
          B <- first half of A
          C <- second half of A
          Return concatenate reverse(c) and reverse(B)
```

Let T(n) be the running time of the algorithm of size n. Write down the recurrence relation for T(n) and solve it by unrolling it.

The divide step takes O(n) time when we explicitly copy it. For the recursion, we recurse on two parts of size n/2 each. The conquer step concatenates the two arrays, which involves copying over at least one of the two, so this takes O(n) time. Solving a base case when n = 1 takes constant time as we just return the element.

$$T(n)=\begin{cases} T(n)=2T(n/2)+O(n) \\ O(1) \end{cases}$$

Using the unrolling approach, this solves to:

$$T(n)=2\times(2T(n/2^2))+O(n/2) \quad ¿2\times T(n/2)+O(n)$$

$$¿c\cdot n+2\cdot n/2+4\cdot c\cdot n/4+\ldots=O(nlogn)$$

Problem 3. Given an array *A* holding *n* objects, we want to test whether there is a *majority* element; that is, we want to know whether there is an object that appears in more than *n*/2 positions of *A*.

Assume we can test equality of two objects in $O(1)$ time, but we cannot use a dictionary indexed by the objects. Your task is to design an $O(n \log n)$ time algorithm for solving the majority problem.

a) Show that if *x* is a majority element in the array then *x* is a majority element in the first half of the array or the second half of the array

If x is a majority element in the original array, then, by the pigeon-hole principle, x must be a majority element in at least one of the two halves. Suppose that n is even. If x is a majority element then at least n/2 + 1 entries equal x. By the pigeonhole principle either the first half or the second half must contain n/4 + 1 copies of x, which makes x a majority element within that half.

b) Show how to check in $O(n)$ time if a candidate element x is indeed a majority element.

We can scan the array by counting how many elements equal x. If the count is more than n/2 we declare x to be a majority element. The algorithm scans the array and spends O(1) time per element, so O(n) time overall.

c) Put these observations together to design a divide and conquer algorithm whose running time obeys the recurrence $T(n) = 2T(n/2) + O(n)$

We first break the input array A into two halves, recursively call the algorithm on each half, and then test in A if either of the elements returned by the recursive call is indeed a majority element of A. If that is the case, we return the majority element. Else, we report there is no majority element.

In terms of correctness, we see that if there is no majority element then the algorithm must return 'no majority element' since no matter what the recursive call returns, we always check if an element is indeed a majority element. Otherwise, if there is a majority element x in A we are guaranteed that one of our recursive calls will identify x, by the pigeon hole principle, and the subsequent check will lead the algorithm to return x.

Regarding time complexity, breaking the main problem into two subproblems and testing the two candidate majority elements can be done in O(n) time. Thus, we get

$$T(n)=\begin{cases} T(n)=2T(n/2)+O(n) \\ O(1) \end{cases}$$

d) Solve the recurrence by unrolling it.

$$T(n)=c\cdot n+2\cdot n/2+4\cdot c\cdot n/4+\ldots=O(nlogn)$$

Problem 3. Let A be an array with n distinct numbers. We say that two indices $0\le i<j<n$ form an inversion if A[i] > A[j]. Modify merge sort so that it computers the number of inversions of A.

We augment the standard merge sort algorithm so that in addition to returning the sorted input array, it also returns the number of inversions.

Let L be the sorted left half of the input array and R be the sorted right half of the input array. We can count on the recursive calls to count the inversions within L and within R, but we need to modify the merge procedure to count the number of inversion (i, j) where i is in the first half and j is in the second half.

For each element e in R, let L(e) be the number of elements in L that are greater than e. Note that the number of inversions (i, j) where i is in the first half of the input array and j is in the second half of the array is equal to the sum of L(e).

Furthermore, L(e) can be computed on the fly when e is added to the merged array since elements that remain in L at that time are precisely those elements of L that are greater than e.

```
Def merge-and-count(L, R):
     result <- new array of length |L| + |R|
     count <- 0
     l, r <- 0, 0
     while l + r < |result|:
          index <- 1 + r
          if r >= |R| or (1<|L| and L[1] <R[r]):
               result[index] <- L[1]
               l <- l + 1
          else
               result[index] <- R[r]
               count <- count + |L| - 1
     return result, count

Def count-inversions(A):
     if |A| = 1:
          return A, 0
     else
          B <- first half of A
          C <- second half of A
          sortedB, countB <- count-inversions(B)
          sortedC, countC <- count-inversions(C)
          sortedBC, countBC <- count-inversions(BC)
          return sortedBC, countB + countC + countBC
```

We focus on the correctness of computing the inversions, since the correctness of sorting was argued during the lecture. The correctness can be proven using a short inductive argument on n, the size of A:

Our base case occurs when n = 1, in which case no inversions can occur, returning 0.

Our inductive hypothesis assumes that for all arrays of size k, the algorithm computes the number of inversions correctly. We now show that the algorithm computes the number of inversions for k + 1 correctly as follows:

It splits the array in two halves, each of size at most k, so by our induction hypothesis, the number of inversions in them is computed correctly. Thus, if we can show that we correctly compute the number of inversions having one element in L and one in R, the correctness of our algorithm follows for k + 1. Fortunately, this is indeed the case, as the number of inversions is exactly the number of unprocessed elements in L (ie |L| - l) when we put an element of R in the sorted array.

Problem 4. Given a sorted array A containing distinct non-negative integers, find the smallest integer that isn't stored in the array. For simplicity, you can assume there is such an integer ie A[n-1] > n-1
Example: A = [0, 1, 3, 5, 7], result = 2.

Our approach is very similar to binary search: we consider the middle element, determine which half contains a missing element and recurse on that. So the main question is how to determine whether a half contains a missing element.

This can be done by checking if the integer stored in A[i] = i. If A[i] = i, we know that

there can be no missing values in the first i elements, since all integers are distinct and non negative. If A[i] > i, this means that there is some value missing and hence, we can recurse on the first half.

```
Def find-missing-in-range(A, low, high):
     if low = high then
          return low
     else
          mid <- floor((low + high)/2)
          if A[mid] = mid then
               return find-missing-in-range(A, mid+1, high)
          else
               return find-missing-in-range(A, 0, n-1)
```

Correctness follows directly from the fact that the integers are sorted and distinct, as well as that it's given that there is actually a missing integer. To argue that we indeed recurse on the correct half of the array, we note that we maintain the invariant that we recurse on the half that is missing the smallest element (implied by our check above as to whether A[i] = i). Since the size of the array that we recurse on decreases with each iteration, this implies that we eventually reach an array of size one, which thus must contain the smallest missing element.

The running time analysis is the same as that for binary search. The recursion is

$$T(n)=\begin{cases} T(n)=T(n/2)+O(1) \\ O(1) \end{cases}$$

Which solves to be O(logn).

Divide and Conquer II Problems

Problem 1.

Randomisation

Assignment Problems

Problem 1.1.

Using O-notation, upperbound the running time of the following algorithm, where A is an array containing n integers.

```
1: function ALGORITHM(A)
2:     result ← 1
3:     for i ← 0; i < n; i++ do
4:         for j ← 0; j < 25; j++ do
5:             result ← result * A[i]
6:     return result
```

The function Algorithm(A) takes an input of array A with n elements. It contains two for loops, with the outer loop iterating n times, and the inner loop iterating a fixed 25 times. Lines 2, 4, 5 and 6 take linear time (as the inner for loop iterates exactly 25 times)
Thus, the worst case running time is O(n).

Problem 1.2.

We want to design a list-like data structure that supports the following operations on integers.

- ADDTOFRONT(e): Adds a new element e at the front of the list.
- ADDTOBACK(e): Adds a new element e at the end of the list.
- REMOVEFROMFRONT(): Removes the first element from the list and returns it.
- REMOVEFROMBACK(): Removes the last element from the list and returns it.
- SETELEMENT(i, e): Sets the element of the i-th element of the list to e.

Each operation should run in $O(1)$ time. You can assume that we know that the list will never contain more than k elements (k is **not** a constant and should **not** show up in your running time).

Example execution:

ADDTOBACK(0)	[0]
ADDTOFRONT(-1)	[-1,0]
ADDTOFRONT(2)	[2,-1,0]
SETELEMENT(1,6)	[2,6,0]
REMOVEFROMBACK()	[2,6], returns 0
REMOVEFROMFRONT()	[6], returns 2

(a) Since we need to supporting setting the element of arbitrary positions/ indices in constant time, we'll use an array A. As the list will have length at most k, we'll use an array of length k. To allow us to insert both ends of the array, we'll use a cyclic scheme similar to what we saw for queues. We keep track of two variables start and size, both of which are initially 0.
When we insert or remove at the end of the list, we only need to update size. If we insert at the front, we need to update both start and size: inserting at the front requires us to insert before the current first element, which we can do by moving start one index (modulo k) and inserting the element at the new starting index of the list. We update size to ensure we know where the list ends and how many elements we're storing. Setting an element now becomes a question of computing the correct index in the array and setting the corresponding element.

```
Function addToFront(e):
      start <- start - 1 mod k
      A[start] <- e
      Size <- size + 1

Function addToBack(e):
      A[start + size mod k] <- e
```

```
      Size <- size + 1

Function removeFromFront():
      if size = 0:
            return 'error'
      e <- A[start]
      start <- start + 1 mod k
      size <- size - 1
      return e

Function removeFromBack():
      If size = 0:
            return 'error'
      e <- A[start + size - 1 mod k]
      size <- size - 1
      return e

Function setElement(i, e):
      if i < 0 or i >= size:
            return 'error'
      A[start + i mod k] <- e
```

(b) To show correctness, we first show that start always contains the starting index in A of our list and that size indeed correctly maintains the size of the list. Together these imply that start + size is the index immediately after the end of the list.
We note that we need to update start only if we add or remove an element from the start of the list. Both of these operations indeed move start forward or backward one position as appropriate and thus start is maintained correctly. Since we increment size when we add an element and decrement it when we remove one, size is indeed maintained correctly.
Now that we know that start and size are correct, we can see that we know where the start and end of the list are at all times. Hence, by inserting and removing elements at those positions (or just before them), the first four operations are correct. Similarly, the i-th element can be found at index start + i mod k, so as long as i is less than size we are accessing a valid element in our list and thus our operation is correct.

(c) All operations perform a constant number of elementary operations (assignments, comparisons, simple math, throwing errors) and hence the running time of each operation is O(1) time.

Problem 1.3.

We are given $n \geq 2$ wireless sensors modelled as points on a 1D line and every point has a distinct location modelled as a positive x-coordinate. These sensors are given as a *sorted* array A. Each wireless sensor has the same broadcast radius r. Two wireless sensors can send messages to each other if their locations are at most distance r apart.

Your task is to design an algorithm that returns all pairs of sensors that can communicate with each other, possibly by having their messages forwarded via some of the intermediate sensors. For full marks your algorithm needs to run in $O(n^2)$ time.

We first observe that there are a quadratic number of potential pairs that need to be reported, so we can't spend too much time finding a single pair. We also observe that A[i] and A[j] (i < j) can communicate if and only if all sensors A[j] and A[l] can communicate with each other. In particular, this means that every pair of consecutive sensors in [i : j] can send messages to eachother.
We will keep track of two indices start and current indicating the range such that every pair of consequence sensors in A[start] and A[current] can send messages to each other. We repeatedly check whether A[current] can communicate with A[current + 1] and if so we report all pairs with the left sensor in [start : current] and the right sensor being current + 1. If A[current] cannot communicate with A[current + 1], we update both start and current to be current + 1 and start a new block of sensors where every sensor can communicate with every other sensor.

```
Function reportCommunication(A, r):
      result <- empty set
      start <- 0
      current <- 0
      while current + 1 < n:
            if A[current + 1] - A[current] <= r:
                  for i <- start; i <= current; i++:
                        result <- result U {i, current + 1}
```

Problem 2.1.

We are given a multiset M of integers, i.e., a set in which every integer can occur multiple times. We want to know if we can assign these integers to two multisets M_1 and M_2 such that the total sum of all integers in M_1 equals the total sum of all integers in M_2.

Example:
When we have $M = \{4,1,1,3,5,4,2\}$, we can construct $M_1 = \{1,2,3,4\}$ and $M_2 = \{1,4,5\}$. The sum of the integers in M_1 equals $1+2+3+4 = 10$, which is the same as the sum of the integers in M_2.

We are given two algorithms for this problem:

WRONGALGORITHM starts by removing two duplicates from M (if any exist) and adding one to each of M_1 and M_2. Next, it repeatedly takes the largest integer x from M and tries to find a multiset M' of integers in $M \setminus \{x\}$ such that the sum of the integers in M' is equal to x. If such an M' is found, it adds x to M_1 and M' to M_2. If it can continue this process until M is empty, it returns true. Otherwise it returns false.

BALANCINGALGORITHM sorts the integers in non-increasing order and adds each integer to the multiset whose sum is smallest when that integer is considered. Once all integers are added to one of the two multisets, it returns whether their sums are equal.

```
1:  function WRONGALGORITHM(M)
2:      M_1, M_2 ← ∅, ∅
3:      while M contains some duplicate integer x do
4:          M ← M \ {x, x}
5:          M_1 ← M_1 ∪ {x}
6:          M_2 ← M_2 ∪ {x}
7:      while M ≠ ∅ do
8:          x ← largest integer in M
9:          M ← M \ {x}
10:         if there exist M' ⊆ M s.t. x = sum of integers in M' then
11:             M ← M \ M'
12:             M_1 ← M_1 ∪ {x}
13:             M_2 ← M_2 ∪ M'
14:         else
15:             return false
16:     return true
```

```
1: function BALANCINGALGORITHM(M)
2:     M1, M2 ← ∅, ∅
3:     Sort M in non-increasing order
4:     for each integer x in M do
5:         if sum of integers in M1 ≤ sum of integers in M2 then
6:             M1 ← M1 ∪ x
7:         else
8:             M2 ← M2 ∪ x
9:     return sum of integers in M1 = sum of integers in M2
```

(a) Show that wrongAlgorithm doesn't always return the correct answer by giving a counterexample.

Using the set M = {1, 2, 3, 4}:

- Firstly, the algorithm searches for duplicates, and there are none in M. So it takes the largest integer, x = 4, and the sum, 1 and 3 → M1 = {4} and M2 = {1, 3}
- The largest next integer is 2, but there are no more integers left which sum to 2. Thus, the algorithm returns false.

The counter example above proves the algorithm to be unsuccessful to producing the correct outcome, which would be M1 = {2, 3} and M2 = {1, 4}.

(b) Argue whether balancingAlgorithm always returns the correct answer.

The algorithm also doesn't work, which can be shown by for example constructing an instance where the two largest integers need to be added to the same set.

For example M = {3, 3, 3, 4, 5}.

- M1 = {5}
- M2 = {4}
- M2 = {4, 3}
- M1 = {5, 3}
- M2 = {4, 3, 2}

As shown above, the algorithm would produce M1 = {5, 3} and M2 = {4, 3, 3}, a false output. However the set should correctly sort into M1 = {5, 4} and M2 = {3, 3, 3}

Problem 2.2.

In the lectures we saw how to construct a priority queue using heaps and lists. It is also possible to construct a priority queue using an AVL tree. Describe how to perform the three main operations of a priority queue using an AVL tree as the underlying data structure. INSERT(k, v) and REMOVE-MIN() should take $O(\log n)$ time and MIN() should run in $O(1)$ time. For simplicity, you can assume that the keys will all be unique.

a) Design a priority queue using an AVL tree as the underlying data structure.

b) Briefly argue the correctness of your operation(s).

c) Analyse the running time of your operation(s).

(a) In order to return the minimum in O(1) time, we're going to maintain a pointer min to the minimum element in the AVL tree. Whenever the tree is empty, this pointer is null and we should return an error. Otherwise we return the (k, v) pointed to.
Inserting a new (k, v) simply invokes the regular insert of the AVL tree and afterwards performs a search for the minimum in the tree: from the root follow left children until we reach a node that doesn't have a left child. The node we end at is the minimum, so we update the min to point at this. Technically, we only need to do this if what we inserted is smaller than the current minimum, but in big O notation the running time will be the same anyway, so we'll just do it in either case.
Removing the minimum performs a normal deletion of the node pointed to by min (after we've made sure to store the current (k, v) pointed to so we can return that). Afterwards, we perform the same search as above to find the new minimum. Note that if the tree is empty, we update min to null instead.

(b) We argue that the node with the minimum key can indeed be found by following the path of left children from the root. This follows from the BST property, which states that: $key(p.left) < key(p) < key(p.right)$. Hence, by following the path of the left children, we continue to decrease the key until we reach a node with no left child. In other words, we reached the minimum.
By calling the above search whenever we insert or remove a node, we ensure that min always points to the minimum key in the tree. Hence, by returning this (k, v) whenever min is called, we correctly support this operation. Similarly, remove min removes this (k, v), thus performing its intended function. Finally, insert performs the standard insert, thus guaranteeing that the new key value pair is stored in the AVL tree.

(c) Min runs in O(1) time, since we just follow a single pointer in order to return the intended pair. Insert performs in the standard O(logn) time insertion and after that traverses a single path whose length is at most that of a root to leaf path. Hence, this takes O(logn) time. Finally, remove min performs the standard O(logn) time deletion and after that traverses a single path whose length is at most that of a root to leaf path. Hence, this takes O(logn) time.

Problem 2.3.

We are running a restaurant and as a new promotion management has decided to advertise how many lunch specials we have that cost at most \$15. Every lunch special consists of one main and one side dish. Our chefs like to try out new dishes on a regular basis, so it'd be great if we could efficiently compute the number of such combinations when needed.

Your task is to design a data structure that supports the following operations, where *name* is the name of a dish and *price* is its price (a positive integer):

- ADDNEWMAINDISH(*name*, *price*): Adds a new main dish called *name* costing *price*.
- ADDNEWSIDEDISH(*name*, *price*): Adds a new side dish called *name* costing *price*.
- REMOVEMAINDISH(*name*): Removes the main dish called *name* and returns its price, if *name* exists.
- REMOVESIDEDISH(*name*): Removes the side dish called *name* and returns its price, if *name* exists.
- COUNTCOMBINATIONS(): Returns the number of combinations of a main dish and a side dish with total price at most \$15.

Each operation should run in $O(\log n)$ time where n is the number of dishes (i.e., mains plus side dishes), except COUNTCOMBINATIONS which should run in $O(1)$ time. Your data structure should take $O(n)$ space.

a) Design a data structure that supports the above operations in the required time.

b) Briefly argue the correctness of your data structure and operations.

c) Analyse the running time of your operations and the total space of the data structure.

(a) Since we need to return the number of combinations in O(1) time, we can't do much else than maintain a variable count that keeps track of how many such combinations our data structure contains (initialised at 0), which we return when count combinations is called.
We also maintain two AVL trees, one for the main dishes and one for the side dishes, that use price as their key and allow for supplicate keys by means of adding a list of the different values if the keys are the same (avoiding AVL rebalancing issues)
    - Every node stores the size of the subtree rooted at it, ie the sum of the sizes of the lists stored in its subtree.
    - This size can be maintained through rotations using only a constant number of extra operations per rotation.
    - Now every time we need to add a new main dish, we first use the standard insert to add it to the AVL tree (either adding a new node or adding it at the front of an existing list)
    - Updating count: search the side dish AVL tree for 15 – price of main dish. If at an internal node v, we have that v.price = 15 – price, we add v.left.size + v.list.size to count and end the recursion.

- If we have v.price < 15 – price, we add v.left.size + v.list.size to count and recurse on the right subtree.

Problem 3.1.

Problem 3.2.

(a) Our greedy algorithm sorts the books by their deadline in non-decreasing order and prints all copies of a book when its print run starts. In order to determine whether a book is printed by its deadline, we keep track of the time when we started print it. The completion time is this starting time plus ci x pi. If all books meet their deadline this way, we return true. Otherwise, we return false.

```
Def canPrintAll(B):
     currentTime <- 0
     sort B in non-decreasing order of deadlines and renumber
the books such that d1 <= d2 <= … <= d(n-1)
     for each book bi in B:
          completionTime <- currentTime + ci * pi
          if completionTime <= di:
               currentTime <- completionTime
          else:
               return false
     return true
```

(b) First of all, if we return true, then the completion time of each print run in our greedy schedule was on or before the deadline and thus there exists a valid schedule.
We argue that if there exists a schedule that completes all books before their deadline, then the schedule that our algorithm uses does so as well. Let an inversion be a pair of copies of books bi and bj such that i < j in our sorted order and the copy of bj if printed before the copy of bi. Let I be the number of inversions in the optimal schedule. Since our schedule completes the full print run of all books in the order or B, our schedule has no inversions.
We first observe that if the optimal schedule contains any inversions, it also contains an inversion between two consecutive print jobs. We use an exchange argument to show that swapping the order of these two consecutive print jobs bj and bi reduces the number of inversions and doesn't cause the schedule to miss any deadlines.
Consider the optimal schedule where bi and bj are swapped, ie, bi now occurs before bj. This decreases the number of inversions by 1, hence it remains to show that this schedule still meets all the deadlines. If neither bi nor bj was the last print of its book, the time at which we finish printing the full run of these books hasn't changed, so we still meet our deadlines.

(c) W

Problem 3.3.

Practice Final Exam

Problem 1.

(a) Lines 2,4 and 5 are simple assignment and comparison operations and therefore run in linear O(1) time. Line 4 is a loop dependent on the length of A, and runs in worst case O(n) time. Thus, the algorithm runs in O(n) time.

(b) Using the proof by unrolling approach:

$T(n)=2T(n/2)+O(n)$ ¿ $2(2T(n/4)+O(n/2))+O(n)$ ¿ $4T(n/4)+2O(n)$

¿ $2(4T(n/2^3)+2O(n/2))+O(n)$ ¿ $8T(n/8)+4O(n/2)+O(n)$ ¿ $8T(n/8)+3O(n)$

$T(n)=c*n+2*c*(n/2)+\ldots=O(nlogn)$

Problem 2.

(a) Compute a minimum spanning tree T of G. List the edges in T.

A connected and undirected graph has N^{n-2} number of different spanning trees possible. Graph G has 6 vertices, so 6^4 possible spanning trees.

Using Kruskal's algorithm approach, the graph edges are sorted with respect to their weight as follows:

- (D, E) = 1
- (C, F) = 2
- (E, F) = 3
- (C, E) = 4
- (B, C) = 5
- (A, D) = 6
- (B, D) = 7
- (A, B) = 10

Starting at (D, E) with the smallest weight:

(E, F) → (F, C) → (C, B)

However, it is noted that (A, D) < (A, B), so we connect (A, D) to (D, E) to maintain the unconnected graph property and obtain the minimum spanning tree.

Problem 3.

(a) The data structure below is created using a vector of vectors, which is appropriate as it

Problem 4. We say that two distinct spanning trees T and S of G are one swap away from each other if $|T \cap S| = n - 2$; that is T and S differ in only one edge.

(a) We are given a connected and undirected graph, G. Let V be the set of all vertices in G. The relation R on V is defined as follows:

- For any vertices a, b in V: if there is a path from a to b with an even number of edges, this is the relation R. We have to prove that R is an equivalence relation. In other words, we have to prove that the relation is reflexive, symmetric and transitive.
- The relation R on set V is reflexive if a R a for any a in V.
- A vertex of