DATA STRUCTURES AND ALGORITHMS

Lecture Notes Compilation and Practice Problems

Author

Nitin Vetcha

Instructor

Professor Chandrasekaran Pandurangan

Computational Data Science

Contents

I

Problems

1 Chip Testing

§1.1 Problem

1 (*Chip Testing Problem*)

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

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

We are required to find an algorithm which finds the number of good chips. A naive algorithm runs in $O(n^2)$ which has been improvised with a divide and conquer algorithm. Later linear algorithms were developed. The current solution presents an $O(n)$ algorithm which satisfies the recurrence relation

$$T(n) = \begin{cases} n-2 & \text{if } n \geq 2 \\ 1 & \text{if } n = 3 \\ 0 & \text{if } n = 2 \end{cases}$$

Note that $T(1)$ is undefined because we cant test just a single chip by any means.

§1.2 Solution

We shall define the set of chips $\{c_1, c_2, \ldots, c_n\}$ as S which satisfies the property that *" The number of good chips in S is strictly greater than the number of bad chips in S "*

Thus if g denotes the number of good chips and b denotes the number of bad chips, then we have (where n is the total number of chips)

$$g + b = n \text{ such that } g > b$$

Here since the required Output is a proper subset of the Input, thus pruning / decremental design is a good choice because no computation is involved and only reasoning / elimination is necessary.

At each step we keep updating a set $S^{'}$ initially initialized to ϕ such that at each step $|S| > |S^{'}|$ and $S^{'}$ has more number of good chips than bad chips.

We now do pairwise testing : $(c_1, c_2), (c_3, c_4), \ldots, \lfloor n/2 \rfloor$ pairs. We see that if n is even, all of the chips have been accounted, while if n is odd, c_n would be left. Thus we consider testing of all pairs (c_{2i-1}, c_{2i}) such that $1 \leq i \leq \lfloor n/2 \rfloor$.

Algorithm GCBC($S, |S|$)

```
1:  n ← |S|
2:  if (n = 2)
3:          return S
4:  if (n = 3)
5:          if ( Test(c1, c2) = (good, good) )
6:                  return c1
7:          else
8:                  return c3
9:  S' = φ
10: for i = 1 to ⌊n/2⌋
11:         if ( Test(c2i−1, c2i) = (good, good) )
12:                 S' = S' ∪ c2i−1
13: if ( (n is odd) and (|S'| is even) )
14:         S' = S' ∪ cn
15: return (GCBC(S', |S'|))
```

Let the number of pairs (c_{2i-1}, c_{2i}) which are both (good, good) be x, both (bad, bad) be y and either (good, bad) or (bad, good) be z. By the construction of $S^{'}$, we have

$$|S^{'}| = x + y^{'} \ , y^{'} \leq y$$

We see that in the Worst Case, even if all (bad, bad) chits were to be tested and returned (good, good), we have

$$|S^{'}|_{\max} = x + y$$

$$(g, b) = \begin{cases} (2x + z,\ 2y + z) & \text{if } n \text{ is even} \\ (2x + z + 1,\ 2y + z) & \text{if } n \text{ is odd and } c_n \text{ is good} \\ (2x + z,\ 2y + z + 1) & \text{if } n \text{ is odd and } c_n \text{ is bad} \end{cases} \tag{1.1}$$

Now adhering to the condition that $g > b$, yields the following relations between x ad y

$$(x, y) = \begin{cases} x > y & \text{if } n \text{ is even} \\ x \geq y & \text{if } n \text{ is odd and } c_n \text{ is good} \\ x > y & \text{if } n \text{ is odd and } c_n \text{ is bad} \end{cases} \tag{1.2}$$

Thus, if n is even, then since $S^{'}$ has x good chips and atmost y bad chips, the property holds. While if n is odd, then we can only conclusively state that $x \geq y$. Depending on $|S^{'}|$ we have two

cases. If $|S^{'}|$ is odd, then $x \neq y$, otherwise $|S^{'}|$ would be $2x$ (or $2y$) which is even. Hence $x > y$, and the property again holds true.

However, if n is odd and $|S^{'}|$ is even, then there is a possibility that $x = y$. This implies that $\{c_1, c_2, \ldots, c_{n-1}\}$ contain equal number of good and bad chips. However, since the initial set S satisfies the property c_n should be a good chip. Thus we add it to $S^{'}$ making $x > y$ i.e., the property holds.

We shall show this operation works even if $x > y$ in the above case (n is odd and $|S^{'}|$ is even). Note that if c_n is good, then adding it doesn't affect the property. However, if if c_n is bad, then we note that since $x > y$ and $x + y = 2k$ (say), then we have $x \geq k + 1$, $y \leq k - 1 \Rightarrow x - y \geq 2$. Thus, the even after the addition of one bad chip i.e., c_n, the property would still hold true.

We see that in the worst case the recursive call would be made after adding c_n, thus the resulting recurrence would be (with initial conditions $T(3) = 1, T(2) = 0$)

$$T(n) = \lfloor n/2 \rfloor + T(\lceil n/2 \rceil))$$

It can now easily be showed by strong induction that $T(n) = n - 2$, $n \geq 2$. Alternatively, a direct substitution into the recurrence yields

$$T(n) = \lfloor n/2 \rfloor + \lceil n/2 \rceil - 2$$

$$\Rightarrow T(n) = n - 2 \blacksquare$$

Thus, using the GOOD_CHIP_BAD_CHIP (GCBC) algorithm, we can obtain the good chip which then can be used for testing to determine whether the remaining chips are good or bad.

Note that in the base case when $n = 2$, then since number of good chips is strictly greater than number of bad chips, therefore both must be good and we return S, while when $n = 3$, if test response is (good, good) then since the property holds true, therefore both of the tested chips are good irrespective of c_3, thereby we can return c_1 or c_2 and if the test is otherwise, then atleast one of the tested chips is bad but since the property holds, there must be atleast two good chips, thus we can say with certainty that c_3 is a good chip while (c_1, c_2) is either (good, bad) or (bad, good) and hence return c_3.

2 Majority Element

§2.1 Problem

2 (*Majority Element Problem*)

Given an array A of length n and an element x, we say that x is the ***majority element*** of A, if its frequency i.e., number of occurrences in A is $> \lfloor n/2 \rfloor + 1$ if n is even or $> \lceil n/2 \rceil$ if n is odd. It can be clearly be seen that A contains at most one majority element. We are required to write an algorithm which, given A as an input, outputs the majority element.

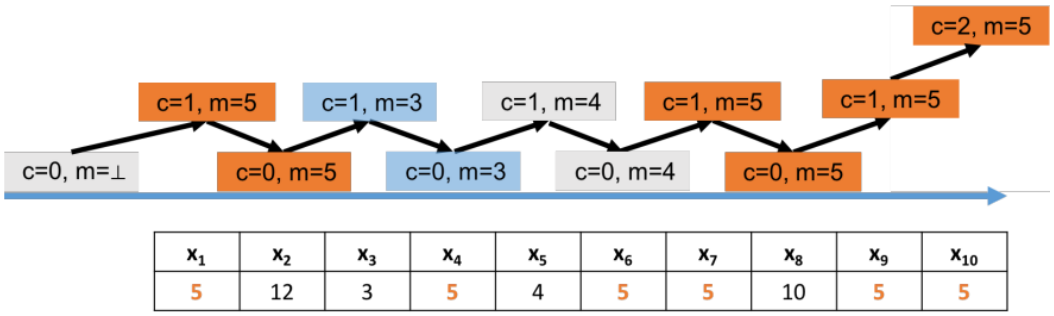
§2.2 Solution

We see a new class of algorithms called **Streaming algorithms** which have several applications in the field of networking where data is transferred in streams of packets. In terms of stream, we can formally restate the majority problem as follows : Let $\sigma =< a_1, \ldots, a_m >$ be a stream of elements and $f = (f_1, \ldots, f_n)$ be the frequency vector of σ. Hence, $\sum_{i=1}^{n} f_i = m$. Then if there exists j such $f_j > m/2$ then output j ; else output ϕ.

Algorithm MAJORITY_ELEMENT(A)

```
1:  m ← ⊥
2:  c = 0
3:  for (i = 1 to n)
4:          if (c = 0)
5:                  m = A[i]
6:                  c = 1
7:          elif (A[i] = m)
8:                  c = c + 1
9:          else
10:                 c = c − 1
11: return m
```

However, in the above algorithm, m returned is only a potential majority element i.e., it may not be actually a majority element. The following diagram indicates how the algorithm updates the values of c and m after each iteration on an an example.

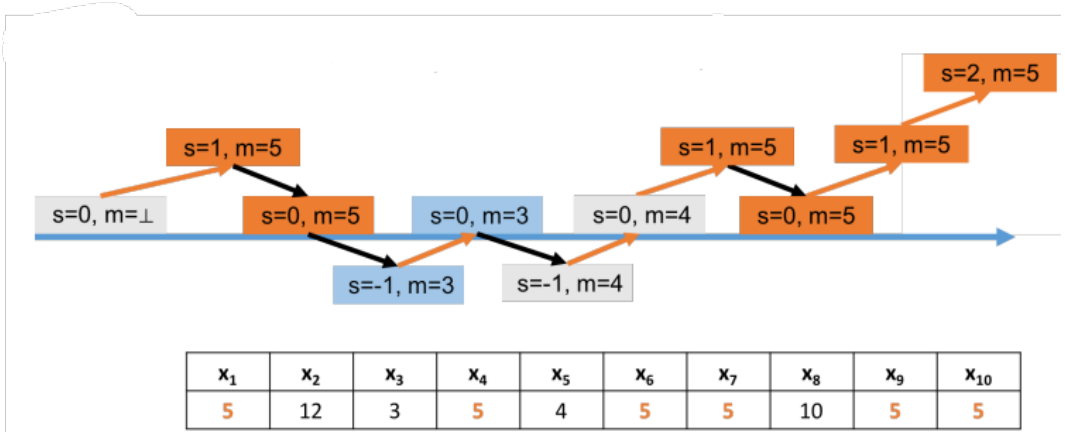


x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
5	12	3	5	4	5	5	10	5	5

Here, we present an analysis of the algorithm which wasn't covered as a part of the lecture.

Claim 1. MAJORITY_ELEMENT algorithm always outputs the majority element, regardless of what order the stream is presented in, assuming it exists

Proof : Let M be the true majority element. Let $s = c$ when $m = M$ and $s = -c$ otherwise (s is a 'helper' variable). s is incremented each time M appears. So it is incremented more than it is decremented (since M appears a majority of times) and ends at a positive value. $\Rightarrow$ **algorithm ends with** $m = M$**.**



x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
5	12	3	5	4	5	5	10	5	5

The above presented algorithm looks deceptively incremental but in practice it is decremental. You can look up further and read about the reservoir problem.

3 Horner's Polynomial Evaluation

§3.1 Problem

3 (*Horner's Polynomial Evaluation Problem*)

The following code fragment implements Horner's rule for evaluating a polynomial given the coefficients $a_0, a_1, \ldots, a_n$ and a value for x.

$$P(x) = \sum_{k=0}^{n} a_n x^k = a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + xa_n)\ldots))$$

Algorithm HORNER'S_RULE($(a_0, a_1, \ldots, a_n), x$)

1: $y \leftarrow 0$
2: **for** ($i = n$ downto 0)
3: $\quad y = a_i + x.y$
4: **return** y

We are required to prove that the given code fragment correctly evaluates a polynomial characterized by the coefficients $(a_0, a_1, \ldots, a_n)$ at the value x.

§3.2 Solution

We shall use loop invariants to prove its correctness. We note that code is a dynamic system in which the state of variables keep changing. Loop invariant is a relation among the variables which remains unchanged after each iteration, thus it cant be a proposition. (At industry grade level, predicate calculus and Z specification language is used to show formal correctness). The approach we adopt is similar to pseudo-predicate calculus.

A common error is to assume that y is a polynomial and make loop invariants like degree(y) $\leq n$, which even though true is not of much helpful. It should be noted that y is a number / value which keeps updating after each iteration and x is a constant. Hence, i and y are the only two variables involved. Thus, if at a particular instance the value of variables i and y are j and α respectively and after some iterations, the values become $j^{'}$ and $\alpha^{'}$ respectively, then loop invariant gives a relation between j and α which holds true for $j^{'}$ and $\alpha^{'}$ as well. Instead of stating the loop invariant repetitively in terms of values, we can use the variable names instead directly. We note that the state of algorithm at the end of iteration No.1 is same as that at the start of iteration No.2, similarly for Iteration No.'s 3 and 4 and so on.

Description	i	y
Start of iteration No.1	n	0
Start of iteration No.2	$n-1$	a_n
Start of iteration No.3	$n-2$	$a_n x + a_{n-1}$
Start of iteration No.4	$n-3$	$a_n x^2 + a_{n-1}x + a_n$
$\vdots$	$\vdots$	$\vdots$
Start of $(n-j+1)^{\text{th}}$ iteration	$n-(n-j) = j$	$a_n x^{(n-j)-1} + \cdots + a_{j+1}$
Start of $(n-j+2)^{\text{th}}$ iteration	$n-(n-j+1) = j-1$	$a_n x^{(n-j)} + \cdots + a_j$
$\vdots$	$\vdots$	$\vdots$
Start of n^{th} iteration	0	$a_n x^{n-1} + \cdots + a_1$
End of n^{th} iteration	-1	$a_n x^n + \cdots + a_1 x + a_0$

Observing the relation between i and y, we state claim that the following relation is a loop invariant.

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

Note that loop invariant generates a relation for an infinite number of steps. However, for correctness, we need in addition to the invariant, information about the algorithm at the time of termination. We can see that $i = -1$ is the termination condition and at this instant $y = \sum_{k=0}^{n} a_k.x^k$ which is exactly the value expected to be computed i.e., value of the polynomial evaluated at x and since the algorithm returns this very value, it concludes the proof of correctness. Lets introduce a dummy variable j and observe the state of the algorithm when $i = j$ in which case the loop invariant states that $y = \sum_{k=0}^{(n-j)-1} a_{j+k+1}.x^k$. Now, we have to show that even when i becomes $i^{'}$ and y becomes $y^{'}$, the loop invariant still holds true. At the start of the next iteration, we have $i = j-1$ and $y^{'}$ as follows : $y^{'} = a_j + x.y = a_j + x(a_{j+1} + a_{j+2}x + \cdots + a_n x^{n-j-1})$

$$\Rightarrow y^{'} = a_j + a_{j+1}x + a_{j+2}x^2 + \cdots + a_n x^{n-j} = \sum_{k=0}^{n-j} a_{j+k}.x^k$$

Moreover, when $i = j-1$, the loop invariant states $y^{'} = \sum_{k=0}^{n-(j-1)-1} a_{j-1+k+1}.x^k = \sum_{k=0}^{n-j} a_{j+k}.x^k$. Since, the above two expressions are same, the loop invariant using dash notation (') correctly represents the state of the algorithm at each stage and consequently at termination as well.

Algorithm HORNER'S_RULE$((a_0, a_1, \ldots, a_n), x)$

1: $y \leftarrow a_n$
2: **for** (i = 1 to n)
3: $\quad y = a_{n-i} + x.y$
4: **return** y

The above algorithm in principle is similar to the earlier version and in fact, the value of y after each iteration is same as well. However, the loop invariant doesn't remain the same. Infact it is $y = \sum_{k=0}^{i-1} a_{n-i+1+k}x^k$.

II

Data Structures

4 Leftist Heap

§4.1 Introduction

We often encounter the problem of merging two heaps. For example, if one of the printers crashes for some reason, then we have to merge the priority queues PQ of the two printers. Before we proceed, we define the following terminology :

$meld(H_1, H_2)$: Return new heap with keys from H_1 and H_2, destroying the heaps H_1 and H_2

An abstract about Priority Queues : $PQ^{'}$ is an abstract data type (ADT) which primarily supports two operations - insert (x, S) and DeleteMax (S). If we use an implementation, wherein a new element is inserted at random, then insert operation takes $O(1)$ time while DeleteMax would take $O(n)$ time. However, if at the time of insertion itself, we place the elements is a sorted manner, then insert operation takes $O(n)$ time while DeleteMax would take $O(1)$ time. Thus, irrespective of the method implementation, the total cost for n insertions and n deletions would be $O(n^2)$.

§4.2 Problem

4 (*Leftist Heap*)

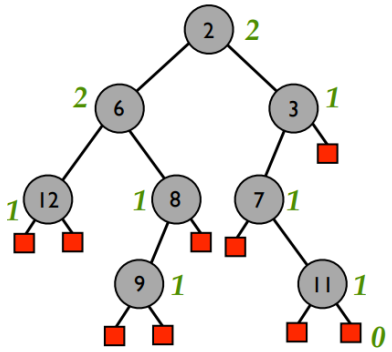
WLOG, assume that we have two $PQ^{'}$s represented using heaps, say PQ_1 (H_1) and $PQ_2(H_2)$ of same size, n. If not we consider n to be the size of the larger priority queue. Then can $meld(H_1, H_2)$ operation be performed in $O(\log n)$ time. To solve this problem, idea of imbalance was exploited and the ***leftist heap*** data structure was developed following a classroom discussion in MIT.

§4.3 Solution

We first introduce another new term ***null path length*** (NPL). NPL of a node is defined as the length of the shortest path from the given node to a leaf. It is defined as follows :

$$\text{NPL}(x) = \begin{cases} 0 & \text{if } x \text{ is an external node / leaf} \\ 1 + \min[\text{NPL}(\text{left}(x)), \text{NPL}(\text{right}(x))] & \text{otherwise} \end{cases} \tag{4.1}$$

Here is an example of a tree wherein the numbers written in green represent the NPL of that node.



Thus, NPL, shortest distance to get to a null pointer, is a value associated with each node describing a property of its subtrees. A leftist heap is a tree with keys in heap order such that NPL(left(x))> NPL(right(x)). The advantage is that if the property is violated at any node, then just swap left and right children at that node.

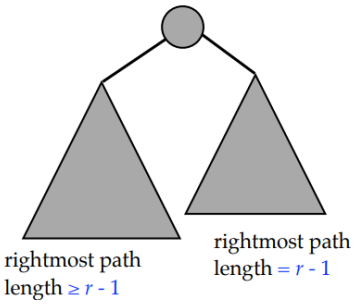
If r denotes the number of nodes in the rightmost path of a leftist heap, then we make two claims.

(a) It has to be complete until the r^{th} level

(b) If n is the total number of nodes, then $n \geq 2^r - 1 \Rightarrow r \leq \log(n+1)$

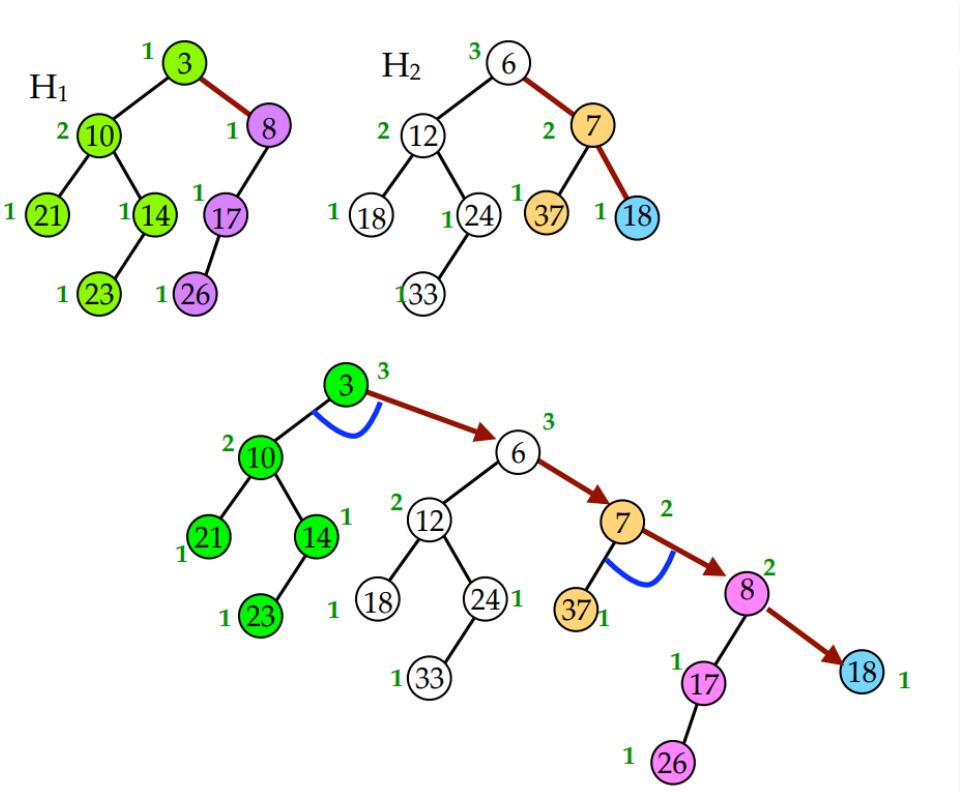
A Proof by contradiction can given for (a) wherein if we suppose that the tree were not complete until the r^{th} level, then there would be a violation of the property at the common ancestor.[*]

We give a proof by strong induction for (b). The base case is $r = 1$, which states $n \geq 2^1 - 1 = 1$ which is true. For the inductive hypothesis, assume $n(i) > 2^i - 1$ for $i < r$. In the induction step, we see that the left and right subtrees of the root, each have atleast $2^{r-1} - 1$ nodes. Thus, the original leftist heap has atleast $2(2^{r-1} - 1) + 1 = 2^r - 1$ nodes. Hence $n \geq 2^r - 1$ so r is $O(\log n)$. Moreover, this reveals the fact that a leftist heap has a shortest path which is the rightmost path using which we will now see how to perform the much-awaited *meld* operation in $O(\log n)$ time.



To perform $meld(H_1, H_2)$, we make a sorted list of the rightmost elements in both the heaps. This is in fact the rightmost path of the resultant heap H. To obtain the entire tree, we keep attaching

the corresponding left subtree of every node we encounter while traversing the rightmost path. It is evident that *meld* takes $O(\log n)$ time. Here is a worked-out example indicating the NPL values for each node as well. We then check for violations along the rightmost path only, moreover, the resultant heap is also leftist.



Later on, Djikstra algorithm required a new operation *decrease_key*, however with a leftist heap this required $O(n)$ time. We shall look later at an abstract data structure that will perform all the operations in $O(n)$.

5 Union Find

§5.1 Introduction

Applications involving grouping of distinct elements into a collection of dis- joint sets i.e., sets with no elements in common often need to perform two operations in particular: finding the unique set that contains a given element and uniting two sets. These so called disjoint-set data structures arise in determining the connected components of an undirected graph for example in Kruskal's algorithm for minimum spanning trees while adding an edge we have to check if the endpoints of the edge belong to same connected component or not, because if they belong then inclusion of the edge would result in a cycle which is violation.

A disjoint-set data structure maintains a collection $S = \{S_1, \ldots, S_n\}$ of disjoint dynamic sets. To identify each set, choose a representative, which is some member of the set. In some applications, it doesn't matter which member is used as the representative; it matters only that if you ask for the representative of a dynamic set twice without modifying the set between the requests, you get the same answer both times. Other applications may require a pre-specified rule for choosing the representative, such as choosing the smallest member in the set (for a set whose elements can be ordered).

§5.2 Problem

We are required to find a data structure which supports the following three operations :

- MAKE-SET(x) , where x does not already belong to some other set, creates a new set whose only member (and thus representative) is x
- UNION(x, y) unites two disjoint, dynamic sets that contain x and y, say S_x and S_y, into a new set that is the union of these two sets and since the sets in the collection must be disjoint UNION destroys the sets S_x and S_y removing them from the collection S. The representative of the new set in principle can be any member of $S_x \cup S_y$.
- FIND-SET(x) returns a pointer to the representative of the unique set containing x

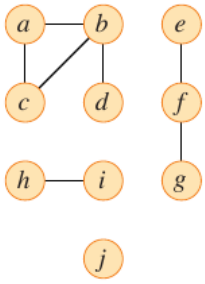
§5.3 Solution

Let G be a graph such that set of vertices is represented by $G.V$ and that of edges by $G.E$. Then the following algorithmic pseudo code determines if two vertices u and v belong to the same connected component or not. In an actual implementation of this connected-components algorithm, the representations of the graph and the disjoint-set data structure would need to reference each other. That is, an object representing a vertex would contain a pointer to the

corresponding disjoint-set object, and vice versa. Since these programming details depend on the implementation language, we address them in the subsequent section.

Algorithm CONNECTED(G, u, v)

```
1:  CONNECTED-COMPONENTS(G)
2:      for (each vertex v ∈ G.V)
3:          MAKE-SET(v)
4:      for (each edge (u, v) ∈ G.E)
5:          if (FIND-SET(u) ≠ FIND-SET(v))
6:              UNION(u, v)
7:
8:  SAME-COMPONENT(u, v)
9:      if (FIND-SET(u) = FIND-SET(v))
10:         return True
11:     else
12:         return False
```



Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,f)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,f\}$		$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,f\}$		$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,f\}$		$\{g\}$	$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,f\}$		$\{g\}$	$\{h,i\}$		$\{j\}$
(f,g)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$
(b,c)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$

§5.4 Implementation

In Kruskal's algorithm, two vertices a and b are said to be related i.e., (aRb) if and only if they belong to the same connected component i.e., there exists a path from a to b. This being an equivalence relation induces a partition on the set of vertices. Thus, the mathematical model for Kruskal's Algorithm is partition which is defined as follows : k^{th} partition on a collection of disjoint sets S produces new sets $\{A_1, A_2, \ldots, A_k\}$ such that $\cup_{i=1}^{k} A_i = S$ i.e., collectively exhaustive and $A_i \cap A_j = \phi \; \forall \; 1 \leq i < j \leq k$. i.e., mutually exclusive. Hence, singleton set is the coarsest partition while $(n-1)^{\text{th}}$ partition is the finest partition and any other partition is intermediate between these two extremes.

§5.4.i Name Array (List-based)

In the discussion below, it will be convenient to define n as the number of MAKE-SET operations and m as the total number of operations (this matches the number of vertices and edges in the graph up to constant factors, and so is a reasonable use of n and m). Our first data structure is a simple one with a very cute analysis. The total cost for the operations will be $O(m + n \log n)$.

In this data structure, the sets will be just represented as linked lists: each element has a pointer to the next element in its list. However, we will augment the list so that each element also has a pointer directly to head of its list. The head of the list is the representative element. We can now implement the operations as follows:

- MAKE-SET(x) : set $x \rightarrow head = x$, takes constant time
- FIND-SET(x) : return $x \rightarrow head$, takes constant time
- UNION(x, y) : To perform a union operation we merge the two lists together, and reset the head pointers on one of the lists to point to the head of the other. Let A be the list containing x and B be the list containing y, with lengths L_A and L_B respectively. Store the length of each list in the head. Then compare and insert the shorter list into the middle of the longer one. Then update the length count to $L_A + L_B$. This takes $O(\min(L_A, L_B))$ time.

The FIND-SET and MAKE-SET operations are constant time so they are covered by the $O(m)$ term. Each UNION operation has cost proportional to the length of the list whose head pointers get updated. So, we need to find some way of analyzing the total cost of the Union operations.

Here is the key idea: we can pay for the union operation by charging $O(1)$ to each element whose head pointer is updated. So, all we need to do is sum up the costs charged to all the elements over the entire course of the algorithm. Let's do this by looking from the point of view of some lowly element x. Over time, how many times does x get walked on and have its head pointer updated? The answer is that its head pointer is updated at most log n times. The reason is that we only update head pointers on the smaller of the two lists being joined, so every time x gets updated, the size of the list it is in at least doubles, and this can happen at most log n times. So, we were able to pay for unions by charging the elements whose head pointers are updated, and no element gets charged more than $O(\log n)$ total, so the total cost for unions is $O(n \log n)$, or $O(m + n \log n)$ for all the operations together.

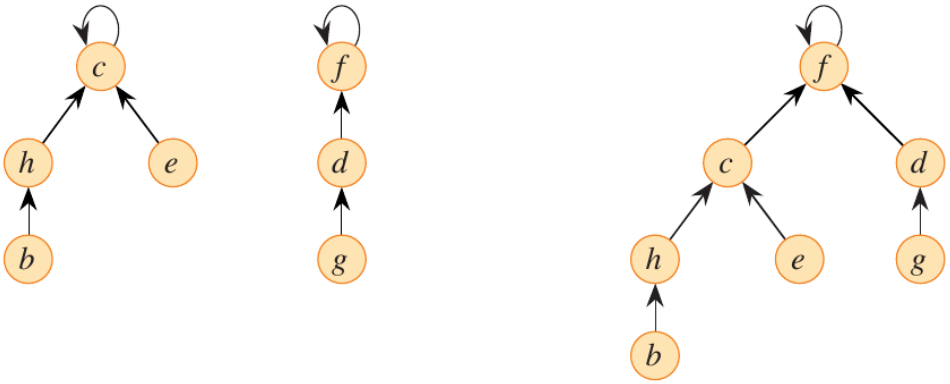
We can alternatively avoid the pointer approach and use the name attribute. An implementation of this approach for Kruskal's algorithm is shown below. Here L is the list of edges sorted in the increasing order of weight represented in the form of endpoints as (u, v) and V is the set of vertices. NEXT(e, L) implies the next edge in the list L. FIND-SET operation takes $O(1)$ time while UNION operation takes $O(n)$ time in worst case scenario. We subsequently see as to how we can improvise this further.

Algorithm Kruskal's Algorithm - NAME ARRAY Implementation (L)

```
1:  for (i = 1 to n)
2:          MAKE-SET(i) // name[i] = i
3:  T ← φ
4:  e ← L[1]
5:  while (not end of L)
6:          e = (x, y)
7:          a = FIND-SET(x)
8:          b = FIND-SET(y)
9:          if (a ≠ b)
10:                 UNION(a, b)
11:                 T = T ∪ {e}
12:         e = NEXT(e, L)
13: return T
```

§5.4.ii Parent Link (Forest-approach)

In a disjoint-set forest, illustrated below, each member points only to its parent. The root of each tree contains the representative and is its own parent. The three disjoint-set operations have simple implementations. A MAKE-SET operation simply creates a tree with just one node. A FIND-SET operation follows parent pointers until it reaches the root of the tree. The nodes visited on this simple path toward the root constitute the find path. A UNION operation, as shown, simply causes the root of one tree to point to the root of the other.



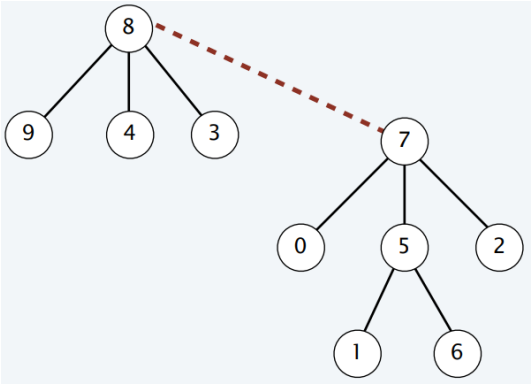
Parent-link representation : Represent each set as a tree of elements.

- Each element has an explicit parent pointer in the tree.
- The root serves as the canonical element (and points to itself).
- FIND-SET(x) finds the root of the tree containing x.
- UNION(x, y) merges trees containing x and y (by making one root point to the other root).

Several optimisations are possible depending on the way two trees are linked while performing the UNION operation. Lets look at them one by one.

5.4.ii.1 Naive Linking

As the name suggests, we naively link the root of first tree to the root of second tree. For example, UNION(5,3) is performed simply as represented in the following figure. Lets look have a closer look at the algorithm and analysis of naive linking.



Algorithm NAIVE_LINKING

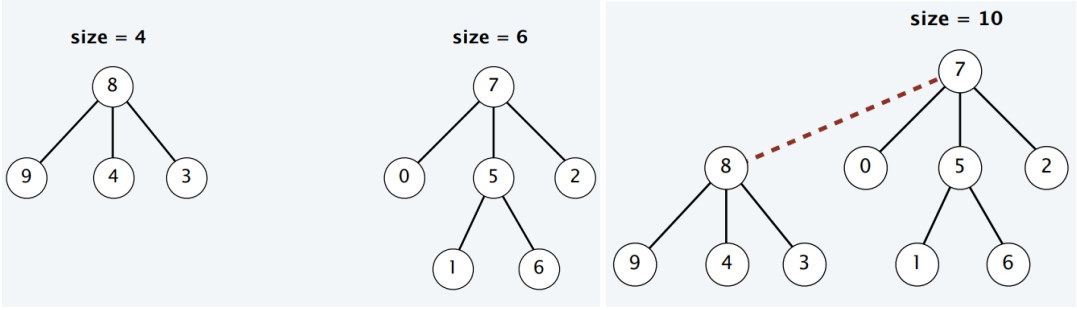
```
1:  MAKE-SET(x)
2:          parent[x] ← x
3:  FIND-SET(x)
4:          while (x ≠ parent[x])
5:                  x ← parent[x]
6:          return x
7:  UNION(x, y)
8:          r ← FIND(x)
9:          s ← FIND(y)
10:         parent[r] ← s
```

Using naive linking, a UNION or FIND operation can take $\Theta(n)$ time in the worst case, where n is the number of elements because in the worst case, FIND takes time proportional to the height (max number of links on any path from root to leaf node) of the tree and height of the tree is n – 1 after the sequence of union operations: UNION(1, 2), UNION(2, 3), . . . , UNION(n – 1, n). Sometimes, linking is also referred to as tree hooking and FIND-SET operation is called ancestor chasing.

5.4.ii.2 Link-by-size

As the name suggests, we maintain a tree size (number of nodes) for each root node and link root of smaller tree to root of larger tree (breaking ties arbitrarily). For example, UNION(5,3) of the

previous case is performed represented in the following figure. Lets look have a closer look at the algorithm and analysis of naive linking but before that we prove that in this manner, for every root node r : size$[r] \geq 2^{\text{height}(r)}$.

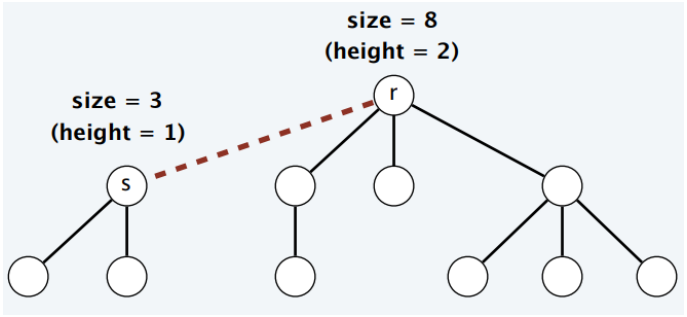


We give a proof by induction on number of links.

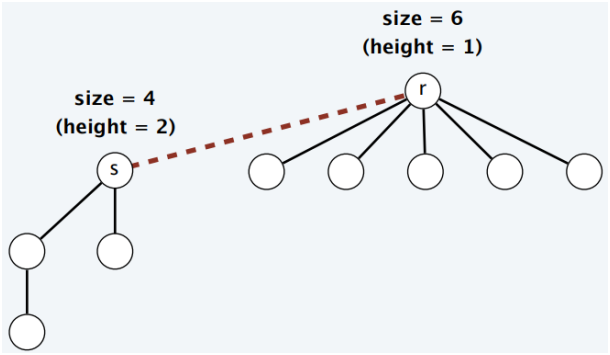
Base Case : We see that when no tree has still been linked i.e., number of links = 0, we have initially a singleton tree which has size 1 and height 0. Since $1 \geq 2^0$, it holds true for base case.

Inductive Hypothesis : Assume true after first i links. Tree rooted at r changes only when a smaller size tree rooted at s is linked into r. Let the resulting tree be r'.

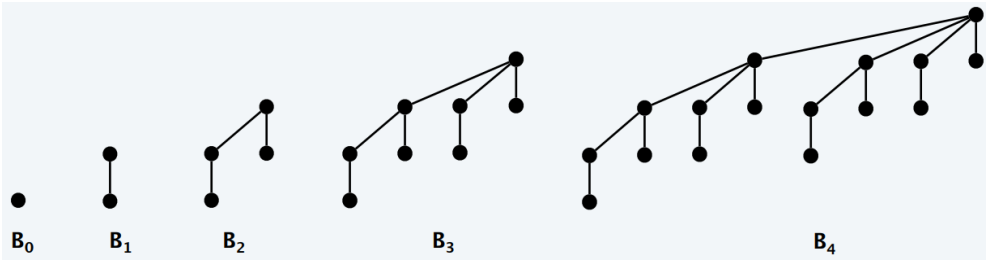
- **Case I :** height$(r) >$ height$(s) \Rightarrow$ size$[r'] >$ size$[r] \geq 2^{\text{height}(r)}$ (by IH) $= 2^{\text{height}(r')}$



- **Case II :** height$(r) \leq$ height$(s) \implies$ size$[r'] =$ size$[r]$ + size$[s] \geq 2.$size$[s]$ (link-by-size) $\geq 2.2^{\text{height}(s)}$ (by IH) $= 2^{\text{height}(s)+1} = 2^{\text{height}(r')}$



Using link-by-size, any UNION or FIND operation takes $O(\log n)$ time in the worst case, where n is the number of elements because the running time of each operation is bounded by the tree height and by previous property, the height $\leq \lfloor \log_2 n \rfloor$. Moreover, we see that this is a tight upper bound because using link-by-size, a tree with n nodes can have height $= \log_2 n$ if we arrange $2^k - 1$ calls to UNION to form a binomial tree of order k where an order-k binomial tree has 2^k nodes and height k.

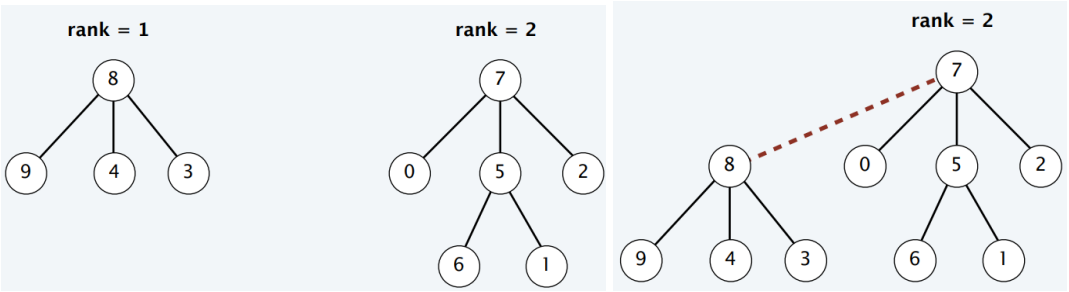


Algorithm LINK_BY_SIZE

```
1:  MAKE-SET(x)
2:          parent[x] ← x
3:          size[x] ← 1
4:  FIND-SET(x)
5:          while (x ≠ parent[x])
6:                  x ← parent[x]
7:          return x
8:  UNION(x, y)
9:          r ← FIND(x)
10:         s ← FIND(y)
11:         if (size[r] > size[s])
12:                 parent[s] ← r
13:                 size[r] = size[r] + size[s]
14:         else
15:                 parent[r] ← s
16:                 size[s] = size[r] + size[s]
```

5.4.ii.3 Link-by-rank

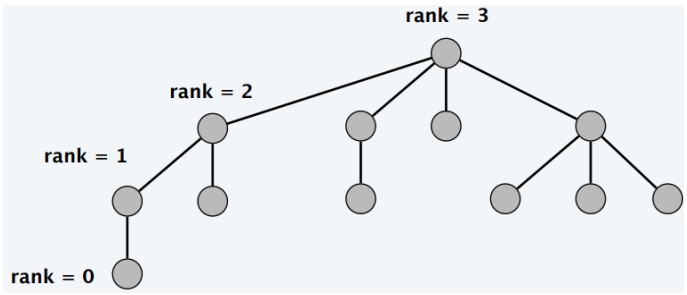
As the name suggest, we maintain an integer rank for each node which is initially 0, the link root of smaller rank to root of larger rank and in case of tie, increase rank of larger root by 1. For now, rank = height. As an example, UNION(5,3) of the previous cases is performed represented in the following figure.



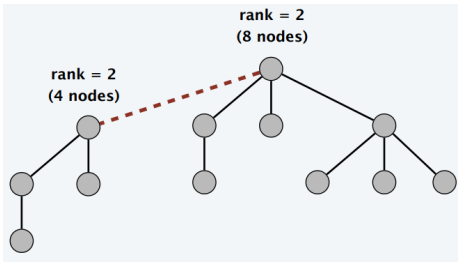
(P1) If x is not a root node, then rank[x] < rank[parent[x]] because a node of rank k is created only by linking two roots of rank $k-1$.

(P2) If x is not a root node, then rank[x] will never change again because rank changes only for roots and a non-root never becomes a root.

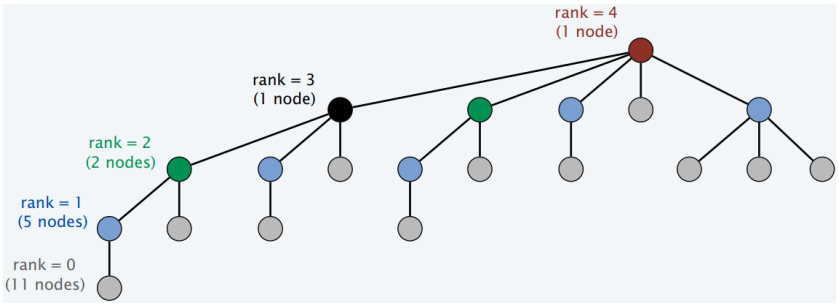
(P3) If parent[x] changes, then rank[parent[x]] strictly increases because the parent can change only for a root, so before linking parent[x] = x. After x is linked-by-rank to new root r we have rank[r] > rank[x].



(P4) Any root node of rank k has $\geq 2^k$ nodes in its tree. We prove this bu induction on k. We see that it is true for $k = 0$ which is the base case. We assume it holds true for $k-1$. Now, a node of rank k is created only by linking two roots of rank $k-1$. By inductive hypothesis, each subtree has $\geq 2^{k\smile}1$ nodes hence the resulting tree has $\geq 2^k$ nodes. Combining this with the first bullet point, we see that the highest rank of any node is $\leq \lfloor \lg n \rfloor$.



(P5) For any integer $k \geq 0$, there are $\leq n/2^k$ nodes with rank k. To see this, note that (P4) implies that any root node of rank k has $\geq 2^k$ descendants. Also any non-root node of rank k has $\geq 2^k$ descendants because by (P4) it had this property just before it became a non-root, also by (P2) its rank doesn't change once it becomes a non-root and its set of descendants doesn't change once it became a non-root. By (P1) different nodes of rank k can't have common descendants.



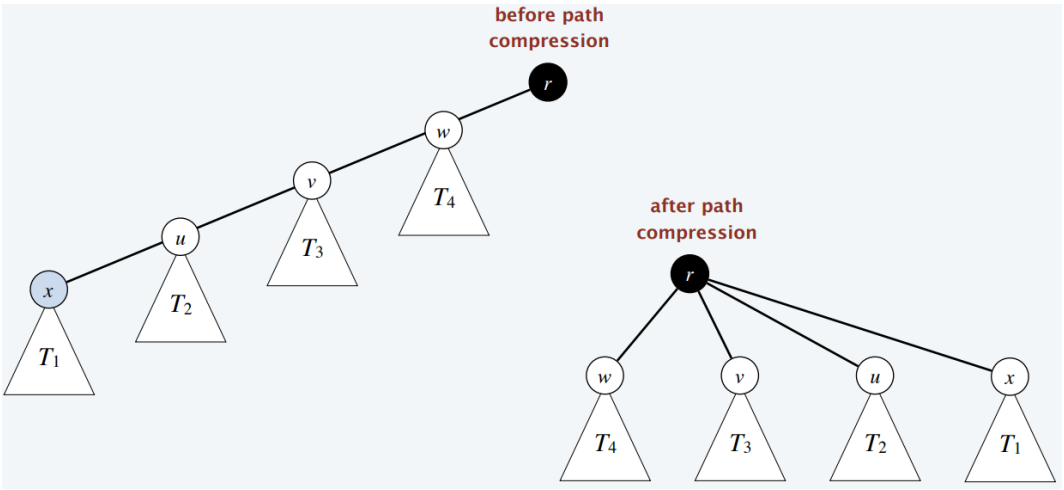
Algorithm LINK_BY_RANK

```
1:  MAKE-SET(x)
2:          parent[x] ← x
3:          rank[x] ← 1
4:  FIND-SET(x)
5:          while (x ≠ parent[x])
6:                  x ← parent[x]
7:          return x
8:  UNION(x, y)
9:          r ← FIND(x)
10:         s ← FIND(y)
11:         if (rank[r] > rank[s])
12:                 parent[s] ← r
13:         if (rank[r] < rank[s])
14:                 parent[r] ← s
15:         else
16:                 parent[r] ← s
17:                 rank[s] = size[s] + 1
```

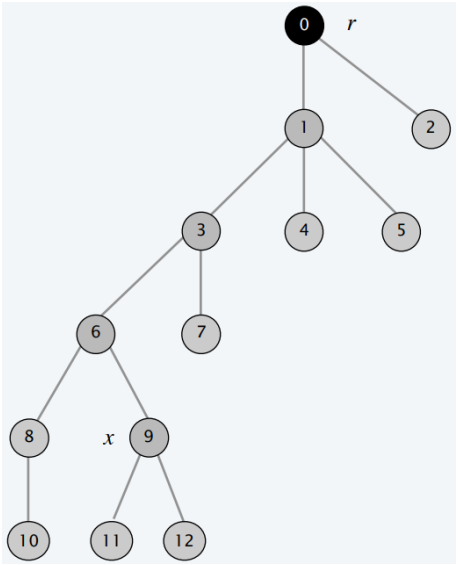
Using link-by-size, any UNION or FIND operation takes $O(\log n)$ time in the worst case, where n is the number of elements because the running time of each operation is bounded by the tree height and by previous property, the height $\leq \lfloor \log_2 n \rfloor$.

5.4.ii.4 Path compression

In this method, while finding the root r of the tree containing x, we change the parent pointer of all nodes along the path to point directly to r.

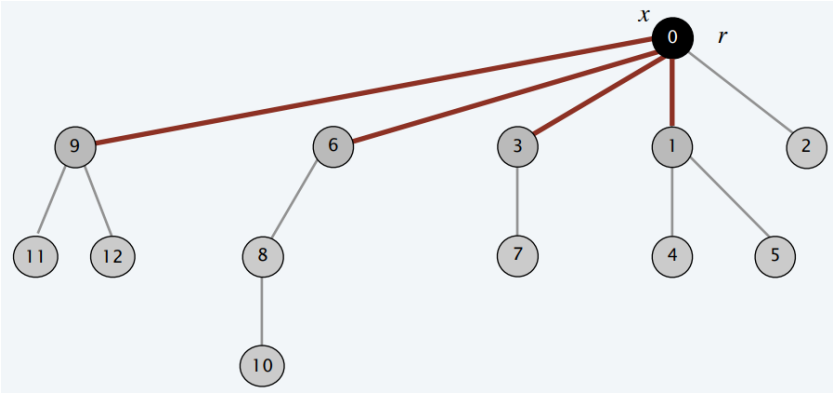


For better understanding of its working, lets look at the following initial tree where the root r of the tree is 0 and we are trying to find the root of x which is 9.



When we apply path-compression technique on this initial tree, we get the resulting final modified version of the tree after a finite number of steps. (When x is at 9, it points the subtree rooted at 9

to 0, then x becomes the root of 9 which is 6 and then points the subtree rooted at 6 to 0, and so on this process continues)



Algorithm PATH_COMPRESSION

```
1: FIND-SET(x)
2:         if (x ≠ parent[x])
3:                 parent[x] ← FIND-SET(parent[x])
4:         return parent[x]
```

Note that by path-compression, the height of the node doesn't get changed. Moreover with naive linking, it takes $\Omega(n)$ time to perform a single UNION or FIND-SET operation, if n is the number of elements because height of the tree is $n-1$ after the sequence of UNION operations - UNION(1, 2), UNION(2, 3), ..., UNION($n-1$, n).

5.4.ii.5 Link-by-Rank with Path compression

$$\lg^* n = \begin{cases} 0 & \text{if } n \leq 1 \\ 1 + \lg^*(\lg n) & \text{otherwise} \end{cases}$$

Let us once revisit the properties of ranks, and state them here formally before we prove that the algorithm based on this method runs in $O(m \lg^* n)$ where $\lg^*$ is the iterated logarithm function defined as above.

n	$\lg^* n$
1	0
2	1
[3, 4]	2
[5, 16]	3
[17, 65536]	4
[65537, 2^{65536}]	5

Moreover, we can see that $\log^* n \leq 5$, unless n itself exceeds the number of atoms in the universe. Hence, for all practical purposes we can say that the algorithm runs in order of $6m$ time. However, if you say that the algorithm is linear to an algorithmic research scientist, you are bound to anger them, because historically it has been an extremely challenging task to find its order.

The bound has been subsequently improved over the years from log to log log to then $\log^*$ and now finally to $O(m\alpha(m,n))$ where $\alpha(m,n)$ a functional inverse of the Ackermann function $A(m,n)$, which grows more slowly than $\log^*$ and is defined as follows :

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$$

$$\alpha(m,n) = \min(i \geq 1 : A(i, \lfloor m/n \rfloor) \geq \log_2 n)$$

Properties of ranks

1. The rank of a node is the same as what the height of its subtree would be if we didn't do path compression. This is easy to see: if you take two trees of different heights and join them by making the root of the shorter tree into a child of the root of the taller tree, the heights do not change, but if the trees were the same height, then the final tree will have its height increase by 1.

2. If x is not a root, then rank(x) is strictly less than the rank of x's parent. We can see this by induction: the UNION operation maintains this property, and the FIND-SET operation only increases the difference between the ranks of nodes and their parents.

3. The rank of a node X can only change if x is a root. Furthermore, once a node becomes a non-root, it is never a root again. These are immediate from the algorithm.

4. There are at most $n/2^r$ nodes of rank $\geq r$. The reason is that when a (root) node first reaches rank r, its tree must have at least 2^r nodes (this is easy to see by induction). Furthermore, by property 2, all the nodes in its tree (except for itself) have rank $< r$, and their ranks are never going to change by property 3. This means that
   (a) for each node x of rank $\geq r$, we can identify a set S_x of at least $2^r - 1$ nodes of smaller rank, and
   (b) for any two nodes x and y of rank $\geq r$, the sets S_x and S_y are disjoint. Since there are n nodes in total, this implies there can be at most $n/2^r$ nodes of rank $\geq r$.

Theorem : *The above tree-based algorithm has total running time $O(m \lg^* n)$*

Proof : Let's begin with the easy parts. First of all, the UNION does two FIND-SET operations plus a constant amount of extra work. So, we only need to worry about the time for the (at most $2m$) Find operations. Second, we can count the cost of a FIND-SET operation by charging \$1 for each parent pointer examined. So, when we do a FIND-SET(x),

(a) if x was a root then pay \$1 (just a constant, so that's ok)

(b) if x was a child of a root we pay \$2 (also just a constant, so that's ok also).

If x was lower, then the very rough idea is that (except for the last \$ 2) every dollar we spend is shrinking the tree because of our path compression, so we'll be able to amortize this cost somehow. For the remaining part of the proof, we're only going to worry about the steps taken in a FIND-SET(x) operation up until we reach the child of the root, since the remainder is just constant time per operation. We'll analyze this using the ranks, and the properties we figured out above.

- **Step 1 :** Let's imagine putting non-root nodes into buckets according to their rank. Bucket 0 contains all non-root nodes of rank 0, bucket 1 has all of rank 1, bucket 2 has ranks 2 through $2^2 - 1$, bucket 3 has ranks 2^2 through $2^{2^2} - 1$, bucket 4 has ranks 2^{2^2} through $2^{2^{2^2}}$ -1, etc. In general, a bucket has ranks r through $2^r - 1$. In total, we have $O(\lg^* n)$ buckets. The point of this definition is that the number of nodes with rank in any given bucket is at most n/(upper bound of bucket + 1), by property (4) of ranks above.

- **Step 2 :** When we walk up the tree in the FIND-SET(x) operation, we need to charge our steps to something. Here's the rule we will use: if the step we take moves us from a node u to a node v such that v is in the same bucket as u, then we charge it to node u (the walk-ee). But if v is in a higher bucket, we charge that step to x (the walk-er).

  The easy part of this is the charge to x. We can move up in buckets at most $O(\lg^* n)$ times, since there are only $O(\lg^* n)$ different buckets. So, the total cost charged to the walk-ers (adding up over the m FIND-SET operations) is at most $O(m \lg^* n)$.

  The harder part is the charge to the walk-ee. The point here is that first of all, the node u charged is not a root, so its rank is never going to change. Secondly, every time we charge this node u, the rank of its new parent (after path compression) is at least 1 larger than the rank of its previous parent by property 2. Now, it is true that increasing the rank of u's parent could conceivably happen log n times, which to us is a "big" number. But, once its parent's rank becomes large enough that it is in the next bucket, we never charge node u again as walk-ee. So, the bottom line is that the maximum charge any node u gets is the range of his bucket.

  To finish the proof, we just said that a bucket of upper bound $B - 1$ has at most n/B elements in it, and its range is $\leq B$. So, the total charge to all walk-ees in this bucket over the entire course of the algorithm is at most $(n/B)B = n$. (Remember that the only elements being charged are non-roots, so once they start getting charged, their rank is fixed so they can't jump to some other bucket and start getting charged there too.) This means the total charge of this kind summed over all buckets is at most n times the number of buckets which is $O(n \lg^* n)$.

6 Priority Queues

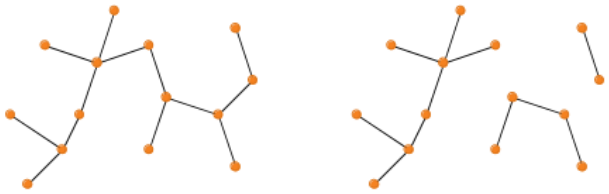
§6.1 Introduction

UNION operation in ordinary binary heaps takes $\Theta(n)$. In attempt to improve the performance in this aspect we shall look at Binary heaps and then subsequently Fibonacci heaps. The operations, supported by all these data structures in general are

- MAKE -HEAP() creates and returns a new heap containing no elements
- INSERT(H, x) inserts node x, whose key field has already been filled in, into heap H
- MINIMUM(H) returns a pointer to the node in heap H whose key is minimum
- EXTRACT-MIN(H) deletes the node from heap H whose key is minimum, returning a pointer to the node
- UNION(H_1, H_2) creates and returns a new heap that contains all the nodes of heaps H_1 and H_2. Heaps H_1 and H_2 are destroyed by this operation
- DECREASE-KEY (H, x, k) assigns to node x within heap H the new key value k, which is assumed to be no greater than its current key value
- DELETE(H, x) deletes node x from heap H

§6.2 Trees

Free Tree A free tree is a connected, acyclic, undirected graph. If an undirected graph is acyclic but possibly disconnected, it is a forest. If $G = (V, E)$ is free tree, then any two vertices in G are connected by a unique simple path. Moreover, if an edge is removed from E, the resulting graph is disconnected but if any edge is added to E, the resulting graph contains a cycle and $|E| = |V| - 1$. In the below figure, (a) is a free tree while (b) is a forest

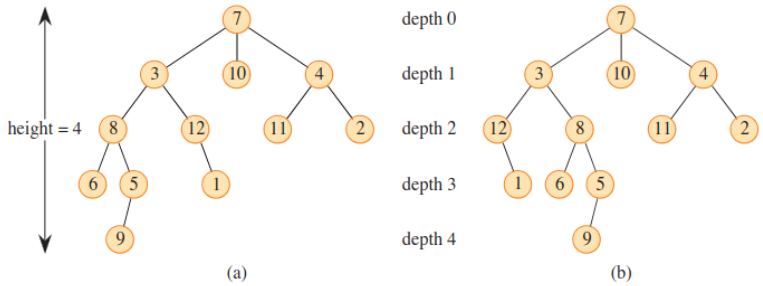


Rooted Tree

A rooted tree is a free tree in which one of the vertices is distinguished from the others. The

distinguished vertex is called the root of the tree. We often refer to a vertex of a rooted tree as a node of the tree. Consider a node x in a rooted tree T with root r. Any node y on the unique path from r to x is called an ancestor of x. If y is an ancestor of x, then x is a descendant of y. (Every node is both an ancestor and a descendant of itself.) If y is an ancestor of x and $x \neq y$, then y is a proper ancestor of x and x is a proper descendant of y. The subtree rooted at x is the tree induced by descendants of x, rooted at x.

If the last edge on the simple path from the root r of a tree T to a node x is (y, x), then y is the parent of x, and x is a child of y. The root is the only node in T with no parent. If two nodes have the same parent, they are siblings. A node with no children is a leaf or external node. A nonleaf node is an internal node.



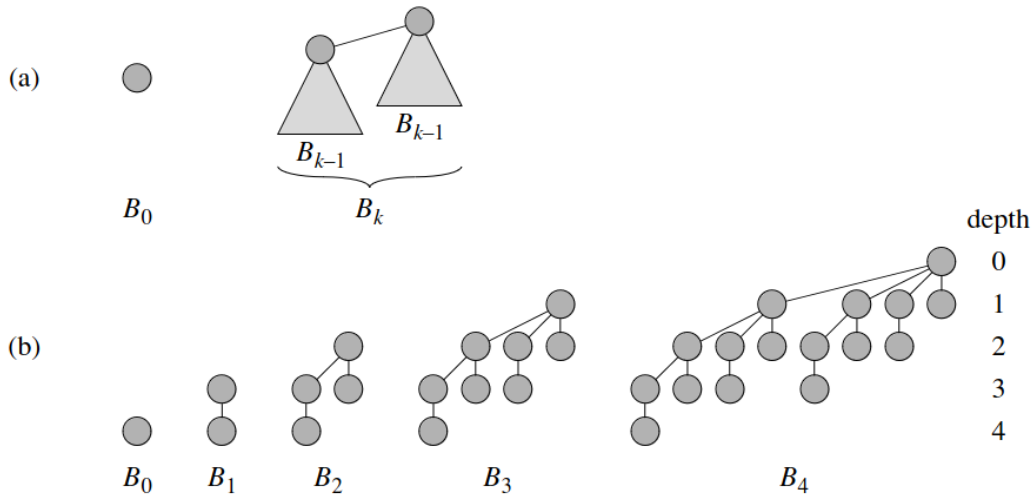
The number of children of a node x in a rooted tree T is the degree of x. The length of the simple path from the root r to a node x is the depth of x in T. A level of a tree consists of all nodes at the same depth. The height of a node in a tree is the number of edges on the longest simple downward path from the node to a leaf, and the height of a tree is the height of its root. The height of a tree is also equal to the largest depth of any node in the tree.

Ordered Tree

An ordered tree is a rooted tree in which the children of each node are ordered. That is, if a node has k children, then there is a first child, a second child, . . . , and a k-th child. In the above figure, (a) is a rooted tree with height 4. The tree is drawn in a standard way: the root (node 7) is at the top, its children (nodes with depth 1) are beneath it, their children (nodes with depth 2) are beneath them, and so forth. If the tree is ordered, the relative left- to-right order of the children of a node matters; otherwise, it doesn't and (b) is another rooted tree. As a rooted tree, it is identical to the tree in (a), but as an ordered tree it is different, since the children of node 3 appear in a different order.

Binomial Tree

The binomial tree B_k is an ordered tree defined recursively. The binomial tree B_0 consists of a single node. The binomial tree B_k consists of two binomial trees B_{k-1} that are linked together - the root of one is the leftmost child of the root of the other.



In this figure, (a) shows the recursive definition of the binomial tree B_k. Triangles represent rooted sub-trees while (b) shows the binomial trees B_0 through B_4.

Lemma 1 (*Properties of Binomial trees*)
For the Binomial tree B_k,

- There are 2^k nodes
- The height of the tree is k
- There are exactly $\binom{k}{i}$ nodes at depth i for $i = 0, 1, \ldots, k$
- The root has degree k, which is greater than that of any other node ; moreover if the children of the root are numbered from left to right by $k-1, k-2, \ldots, 0$, child i is the root of a subtree B_i

Proof : The proof is by induction on k. For each property, the basis is the binomial tree B_0. Verifying that each property holds for B_0 is trivial. For the inductive step, we assume that the lemma holds for B_{k-1}

(1) Binomial tree B_k consists of two copies of B_{k-1} and so B_k has two copies of B_{k-1} and so B_k has $2^{k-1} + 2^{k-1} = 2^k$ nodes.

(2) Because of the way in which the two copies of B_{k-1} are linked to form B_k, the maximum depth of a node in B_k is one greater than the maximum depth in B_{k-1}. By the inductive hypothesis, this maximum depth is $(k-1) + 1 = k$.

(3) Let $D(k, i)$ be the number of nodes at depth i of binomial tree B_k. Since B_k is composed of two copies of B_{k-1} linked together, a node at depth i in B_{k-1} appears in B_k once at depth i and

once at depth $i+1$. In other words, the number of nodes at depth i in B_k is the number of nodes at depth i in B_{k-1} plus the number of nodes at depth $i-1$ in B_{k-1}. Thus,

$$D(k,i) = D(k-1,i) + D(k-1,i-1) = \binom{k-1}{i} + \binom{k-1}{i-1} = \binom{k}{i}$$

(4) The only node with greater degree in B_k than in B_{k-1} is the root, which has one more child than in B_{k-1}. Since the root of B_{k-1} has degree $k-1$, the root of B_k has degree k. Now, by the inductive hypothesis, the children of the root of B_{k-1} are roots of $B_{k-2}, B_{k-3}, \ldots, B_0$. When B_{k-1} is linked to B_{k-1}, therefore, the children of the resulting root are roots of $B_{k-1}, B_{k-2}, \ldots, B_0$.

Remark 1 — The maximum degree of any node in an n-node binomial tree is $\lg n$ which follows immediately from the above properties.

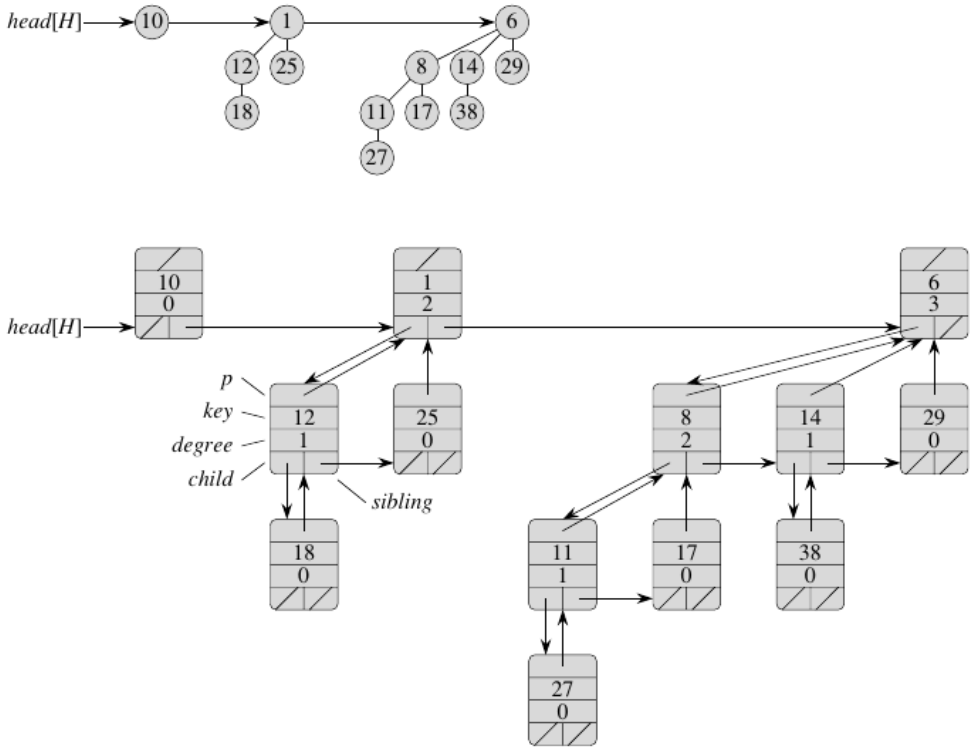
§6.3 Binomial Heaps

Now, a ***Binomial Heap*** H is a set of binomial trees that satisfies the following ***binomial-heap properties***.
(1) Each binomial tree in H obeys the ***min-heap property*** : The key of a node is greater than or equal to the key of its parent. We say that each such tree is ***min-heap-ordered***.
(2) For any non-negative integer k, there is at most one binomial tree in H whose root has degree k.
The first property tells us that the root of a min-heap-ordered tree contains the smallest key in the tree. The second property implies that an n-node binomial heap H consists of at most $\lfloor \lg n \rfloor + 1$ binomial trees. To see why, observe that the binary representation of n has $\lfloor \lg n \rfloor + 1$ bits, say $\langle b_{\lfloor \lg n \rfloor}, b_{\lfloor \lg n \rfloor - 1}, \ldots, b_0 \rangle$ so that $n = \sum_{i=0}^{\lfloor \lg n \rfloor} b_i 2^i$. By property 1 of above Lemma, therefore, binomial tree B_i appears in H if and only if bit $b_i = 1$. Thus, binomial heap H contains at most $\lfloor \lg n \rfloor + 1$ binomial trees.

Representing binomial heaps

Each binomial tree within a binomial heap is stored in a left-child, right-sibling representation. Each node has a key field and any other satellite information required by the application. In addition, each node x contains pointers $p[x]$ to its parent, $child[x]$ to its leftmost child, and $sibling[x]$ to the sibling of x immediately to its right. If node x is a root, then $p[x] = \text{NIL}$. If node x has no children, then $child[x] = \text{NIL}$, and if x is the rightmost child of its parent, then $sibling[x] = \text{NIL}$. Each node x also contains the field $degree[x]$, which is the number of children of x. The roots of the binomial trees within a binomial heap are organized in a linked list, which we refer to as the ***root list***. The degrees of the roots strictly increase as we traverse the root list. By the second binomial-heap property, in an n-node binomial heap the degrees of the roots are a subset of $\{0, 1, \ldots, \lfloor \lg n \rfloor\}$. The *sibling* field has a different meaning for roots than for non-roots. If x is a root, then $sibling[x]$ points to the next root in the root list. (As usual, $sibling[x] = \text{NIL}$ if x is the last root in the root list.) A given binomial heap H is accessed by the field $head[H]$, which is simply a pointer to the first root in the root list of H. If binomial heap H has no elements,

then $head[H]$ = NIL. Below is an example of a binomial heap H consisting of trees B_0, B_2 and B_3, which have 1,4 and 8 nodes respectively, totalling $n = 13$ nodes.



Operations on Binomial Heaps

Creating a new binomial heap

To make an empty binomial heap, the MAKE_BINOMIAL_HEAP procedure simply allocates and returns an object H , where $head[H]$ = NIL . The running time is $\Theta(1)$.

Finding the minimum key

The procedure BINOMIAL_HEAP_MINIMUM returns a pointer to the node with the minimum key in an n-node binomial heap H. This implementation assumes that there are no keys with value ∞. Since a binomial heap is min-heap-ordered, the minimum key must reside in a root node. The BINOMIAL_HEAP_MINIMUM procedure checks all roots, which number at most $\lfloor \lg n \rfloor + 1$, saving the current minimum in min and a pointer to the current minimum in y. When called on the binomial heap shown above, BINOMIAL_HEAP_MINIMUM returns a pointer to the node with key 1. Since, there are at most $\lfloor \lg n \rfloor + 1$ roots to check, the running time of BINOMIAL_HEAP_MINIMUM is $O(\lg n)$.

Algorithm BINOMIAL_HEAP_MINIMUM

```
1: y ← NIL
2: x ← head[H]
3: min ← ∞
4: while (x ≠ NIL)
5:         do if key[x] < min
6:                 then min ← key[x]
7:                         y ← x
8:                 x ← sibling[x]
9: return y
```

Uniting two binomial heaps

The operation of uniting two binomial heaps is used as a subroutine by most of the remaining operations. The BINOMIAL_HEAP_UNION procedure repeatedly links binomial trees whose roots have the same degree. The following procedure links the B_{k-1} tree rooted at node y to the B_{k-1} tree rooted at node z; that is, it makes z the parent of y. Node z thus becomes the root of a B_k tree.

Algorithm BINOMIAL_LINK

```
1: p[y] ← z
2: sibling[y] ← child[z]
3: child[z] ← y
4: degree[z] ← degree[z] + 1
```

The BINOMIAL_LINK procedure makes node y the new head of the linked list of node z 's children in $O(1)$ time. It works because the left-child, right-sibling representation of each binomial tree matches the ordering property of the tree: in a B_k tree, the leftmost child of the root is the root of a B_{k-1} tree. The following procedure unites binomial heaps H_1 and H_2, returning the resulting heap. It destroys the representations of H_1 and H_2 in the process. Besides BINOMIAL_LINK, the procedure uses an auxiliary procedure BINOMIAL_HEAP_MERGE that merges the root lists of H_1 and H_2 into a single linked list that is sorted by degree into monotonically increasing order.

The BINOMIAL_HEAP_UNION procedure has two phases. The first phase, performed by the call of BINOMIAL_HEAP_MERGE , merges the root lists of binomial heaps H_1 and H_2 into a single linked list H that is sorted by degree into monotonically increasing order. There might be as many as two roots (but no more) of each degree, however, so the second phase links roots of equal degree until at most one root remains of each degree. Since the linked list H is sorted by degree, we can perform all the link operations quickly.

In detail, the procedure works as follows. Lines 1–3 start by merging the root lists of binomial heaps H_1 and H_2 into a single root list H. The root lists of H_1 and H_2 are sorted by strictly increasing degree, and BINOMIAL_HEAP_MERGE returns a root list H that is sorted

by monotonically increasing degree. If the root lists of H_1 and H_2 have m roots altogether, BINOMIAL_HEAP_MERGE runs in $O(m)$ time by repeatedly examining the roots at the heads of the two root lists and appending the root with the lower degree to the output root list, removing it from its input root list in the process.

The BINOMIAL_HEAP_UNION procedure next initializes some pointers into the root list of H. First, it simply returns in lines 4–5 if it happens to be uniting two empty binomial heaps. From line 6 on, therefore, we know that H has at least one root. Throughout the procedure, we maintain three pointers into the root list :

- x points to the root currently being examined
- $prev - x$ points to the root preceding x on the root list : $sibling[prev - x] = x$ (since initially x has no predecessor, we start with $prev - x$ set to NIL)
- $next - x$ points to the root following x on the root list : $sibling[x] = next - x$

Algorithm BINOMIAL_HEAP_UNION(H_1, H_2)

```
 1: H ← MAKE_BINOMIAL_HEAP()
 2: head[H] ← BINOMIAL_HEAP_MERGE(H1, H2)
 3: Free the objects H1 and H2 but not the lists they point to
 4: if head[H] = NIL
 5:         return H
 6: prev − x ← NIL
 7: x ← head[H]
 8: next − x ← sibling[x]
 9: while (next − x ≠ NIL)
10:         if ((sibling[next − x] ≠ NIL and degree[siblin[next − x]] = degree[x]) or
11:                                                         (degree[x] ≠ degree[next − x]))
12:                 prev − x ← x
13:                 x ← next − x
14:         else
15:                 if(key[x] ≤ key[next − x])
16:                         sibling[x] ← sibling[next − x]
17:                         BINOMIAL_LINK(next − x, x)
18:                 else
19:                         if (prev − x = NIL)
20:                                 head[H] ← next − x
21:                         else
22:                                 sibling[prev − x] ← next − x
23:                         BINOMIAL_LINK(x, next − x)
24:                         x ← next − x
25:         next − x ← sibling[x]
26: return H
```

Initially, there are at most two roots on the root list H of a given degree since H_1 and H_2 were binomial heaps, they each had at most one root of a given degree. Moreover, BINOMIAL_HEAP_MERGE guarantees us that if two roots in H have the same degree, they are adjacent in the root list. In fact, during the execution of BINOMIAL_HEAP_UNION, there may be three roots of a given degree appearing on the root list H at some time. We shall see in a moment how this situation could occur. At each iteration of the **while** loop of lines 9–25, therefore, we decide whether to link x and $next - x$ based on their degrees and possibly the degree of $sibling[next - x]$. An invariant of the loop is that each time we start the body of the loop, both x and $next - x$ are non-NIL.

Case 1 occurs when $degree[x] \neq degree[next - x]$ i.e., when x is the root of a B_k tree and $next - x$ is the root of a B_l tree for some $l > k$. Lines 12-13 handle this case. We don't link x and $next - x$, so we simply march the pointers one position farther down the list. Updating $next - x$ to point to the node following the new node x is handled in line 25, which is common to every case.

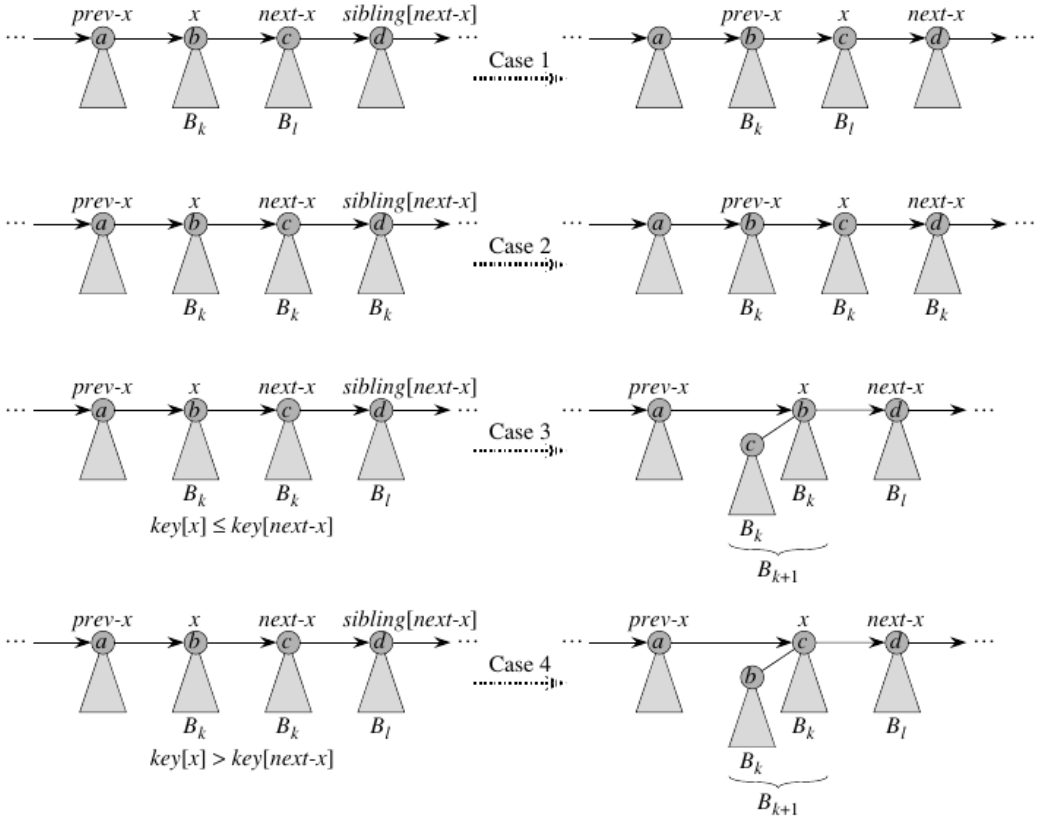
Case 2 is when x is the first of three roots of equal degree, that is, when $degree[x] = degree[next - x] = degree[sibling[next - x]]$. We handle this case in the same manner as case 1 - we just march the pointers one position farther down the list. The next iteration will execute either case 3 or case 4 to combine the second and third of the three equal-degree roots. Line 10-11 tests for both cases 1 and 2, and lines 12–13 handle both cases.

Cases 3 and **4** occur when x is the first of two roots of equal degree, that is, when $degree[x] = degree[next - x] \neq degree[sibling[next - x]]$. These cases may occur in any iteration, but one of them always occurs immediately following case 2. In cases 3 and 4, we link x and $next - x$. The two cases are distinguished by whether x or $next - x$ has the smaller key, which determines the node that will be the root after the two are linked. In case 3, $key[x] \leq key[next - x]$, so $next - x$ is linked to x. Line 16 removes $next - x$ from the root list, and line 17 makes $next - x$ the leftmost child of x. In case 4, $next - x$ has the smaller key, so x is linked to $next - x$. Lines 19–22 remove x from the root list - there are two cases depending on whether x is the first root on the list (line 20) or is not (line 21). Line 23 then makes x the leftmost child of $next - x$, and line 24 updates x for the next iteration.

Following either case 3 or case 4, the setup for the next iteration of the while loop is the same. We have just linked two B_k trees to form a B_{k+1} tree, which x now points to. There were already zero, one, or two other B_{k+1} trees on the root list resulting from BINOMIAL_HEAP_MERGE, so x is now the first of either one, two, or three B_{k+1} trees on the root list. If x is the only one, then we enter case 1 in the next iteration - $degree[x] \neq degree[next - x]$. If x is the first of two, then we enter either case 3 or case 4 in the next iteration. It is when x is the first of three that we enter case 2 in the next iteration.

The running time of BINOMIAL_HEAP_UNION is $O(\lg n)$, where n is the total number of nodes in binomial heaps H_1 and H_2. We can see this as follows. Let H_1 contain n_1 nodes and H_2 contain n_2 nodes, so that $n = n_1 + n_2$. Then, H_1 contains at most $\lfloor \lg n_1 \rfloor + 1$ roots and H_2 contains at most $\lfloor \lg n_2 \rfloor + 1$ and so H contains at most $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2 \leq \lfloor 2 \lg n \rfloor + 2 = O(\lg n)$

roots immediately after the call of BINOMIAL_HEAP_MERGE. The time to perform BINOMIAL_HEAP_MERGE is thus $O(\lg n)$. Each iteration of the while loop takes $O(1)$ time, and there are at most $\lfloor \lg n_1 \rfloor + \lfloor \lg n_2 \rfloor + 2$ iterations because each iteration either advances the pointers one position down the root list of H or removes a root from the root list. The total time is thus $O(\lg n)$.



Inserting a node

Suppose we want to insert a node x into a binomial heap H, then assuming that x has already been allocated and $key[x]$ has already been filled in, we simply make a one-node binomial heap H' in $O(1)$ time (by initialising $p[x]$, $child[x]$, $sibling[x]$ as NIL, $degree[x]$ as 0 and setting $head[H']$ as x) and unite it with the n-node binomial heap H in $O(\lg n)$ time. The call to BINOMIAL_HEAP_UNION takes care of freeing the temporary binomial heap H'.

Extracting the node with minimum key

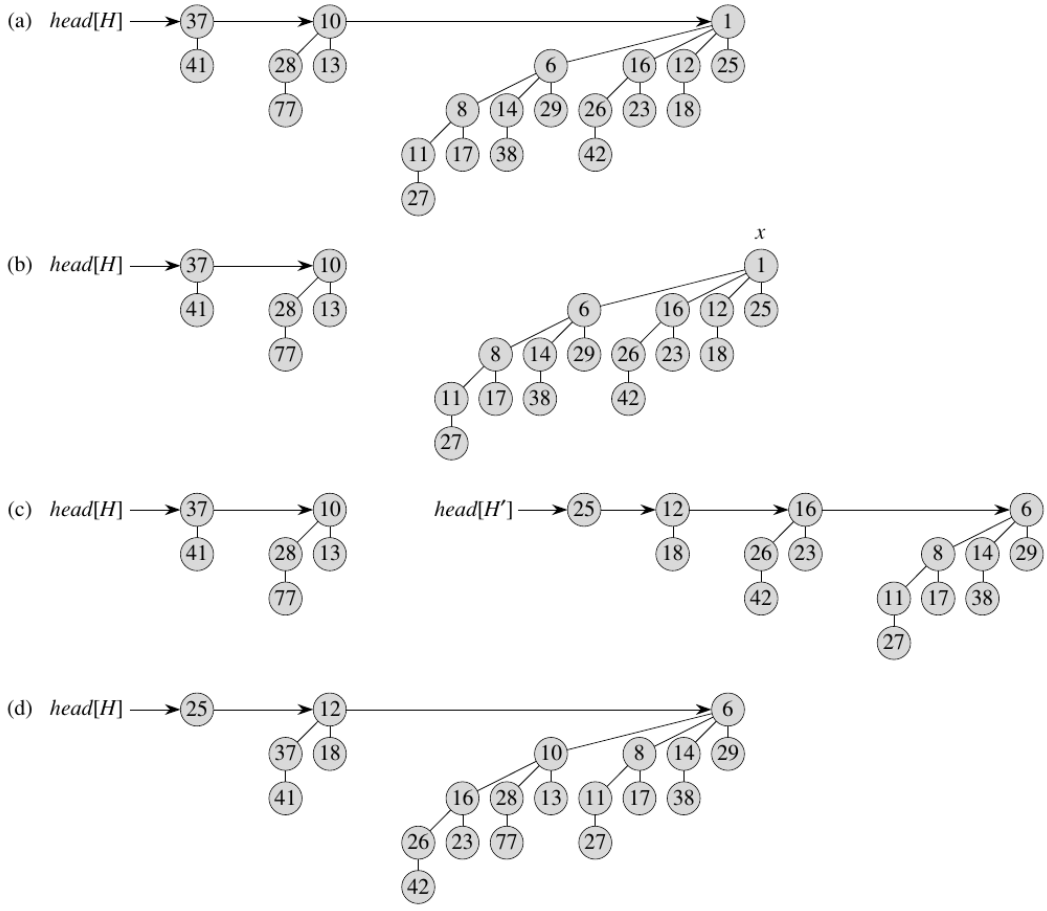
We perform the following procedure, denoted by BINOMIAL_HEAP_EXTRACT_MINIMUM() which extracts the node with the minimum key from binomial heap H and returns a pointer to the extracted node.

- Find the root x with the minimum key in the root list of H, and remove x from the root list

of H

- Create a new binomial heap, say H'. Reverse the order of the linked list of x's children, and set $head[H']$ to point to the head of the resulting list
- Perform BINOMIAL_HEAP_UNION(H, H') and return x

This procedure works since by the last property stated in the lemma, if x is the root of a B_k tree, then its children in order from left to right are roots of $B_{k-1}, B_{k-2}, \ldots, B_0$. Hence, by reversing the list of x's children, we have a binomial heap H' that contains in every node in $x's$ tree except for x itself. Since each of the steps take $O(\lg n)$ time, if H has n nodes, then our procedure takes $O(\lg n)$ time.



In the above figure, we can see the action of BINOMIAL_HEAP_EXTRACT_MIN. (a) is a binomial heap H. (b) shows the root x with minimum key being removed from the root list of H. (c) represents the linked list of x's children in reversed order, giving another binomial heap H'. (d) shows the result of uniting H and H'.

Decreasing a key

The following procedure decreases the key of a node x in a binomial heap H to a new value k. It signals an error if k is greater than x's current key.

Algorithm BINOMIAL_HEAP_DECREASE_KEY(H, x, k)

```
1:  if (k > key[x])
2:  return "error : new key is greater than current key"
3:  key[x] ← k
4:  y ← x
5:  z ← p[y]
6:  while (z ≠ NIL and key[y] < key[z])
7:        exchange key[y] ↔ key[z]
8:        ▷ If y and z have satellite fields, exchange them, too
9:        y ← z
10:       z ← p[y]
```

After ensuring that the new key is in fact no greater than the current key and then assigning the new key to x, the procedure goes up the tree, with y initially pointing to node x. In each iteration of the while loop of lines 6–10, $key[y]$ is checked against the key of y's parent z. If y is the root or $key[y] \geq key[z]$, the binomial tree is now min-heap-ordered. Otherwise, node y violates min-heap ordering, and so its key is exchanged with the key of its parent z, along with any other satellite information. The procedure then sets y to z, going up one level in the tree, and continues with the next iteration. The BINOMIAL_HEAP_DECREASE_KEY procedure takes $O(\lg n)$ time since by property 2 of the lemma, the maximum depth of x is $\lfloor \lg n \rfloor$, so the while loop of lines 6–10 iterates at most $\lfloor \lg n \rfloor$ times.

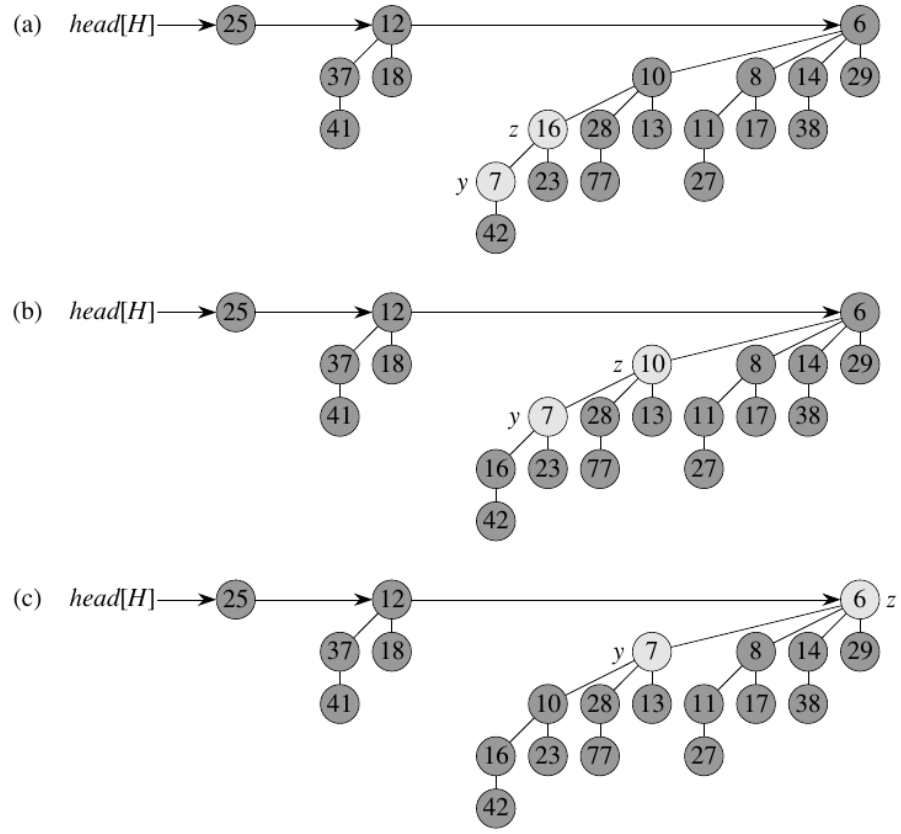
Deleting a key

The following implementation assumes that no node currently in the binomial heap has a key of $-\infty$. It takes $O(\lg n)$ time.

Algorithm BINOMIAL_HEAP_DELETE(H, x)

```
1:  BINOMIAL_HEAP_DECREASE_KEY(H, x, −∞)
2:  BINOMIAL_HEAP_EXTRACT_MIN(H)
```

The below figure shows the action of BINOMIAL_HEAP_DECREASE_KEY. (a) The situation just before line 6 of the first iteration of the while loop. Node y has had its key decreased to 7, which is less than the key of y's parent z. (b) The keys of the two nodes are exchanged, and the situation just before line 6 of the second iteration is shown. Pointers y and z have moved up one level in the tree, but min-heap order is still violated. (c) After another exchange and moving pointers y and z up one more level, we find that min-heap order is satisfied, so the while loop terminates.



§6.4 Fibonacci Heap

Like a binomial heap, a Fibonacci heap is a collection of min-heap-ordered trees. The trees in a Fibonacci heap are not constrained to be binomial trees, however. Unlike trees within binomial heaps, which are ordered, trees within Fibonacci heaps are rooted but unordered. Each node x contains a pointer $p[x]$ to its parent and a pointer $child[x]$ to any one of its children. The children of x are linked together in a circular, doubly linked list, which we call the child list of x. Each child y in a child list has pointers $left[y]$ and $right[y]$ that point to y's left and right siblings, respectively. If node y is an only child, then $left[y] = right[y] = y$. The order in which siblings appear in a child list is arbitrary.

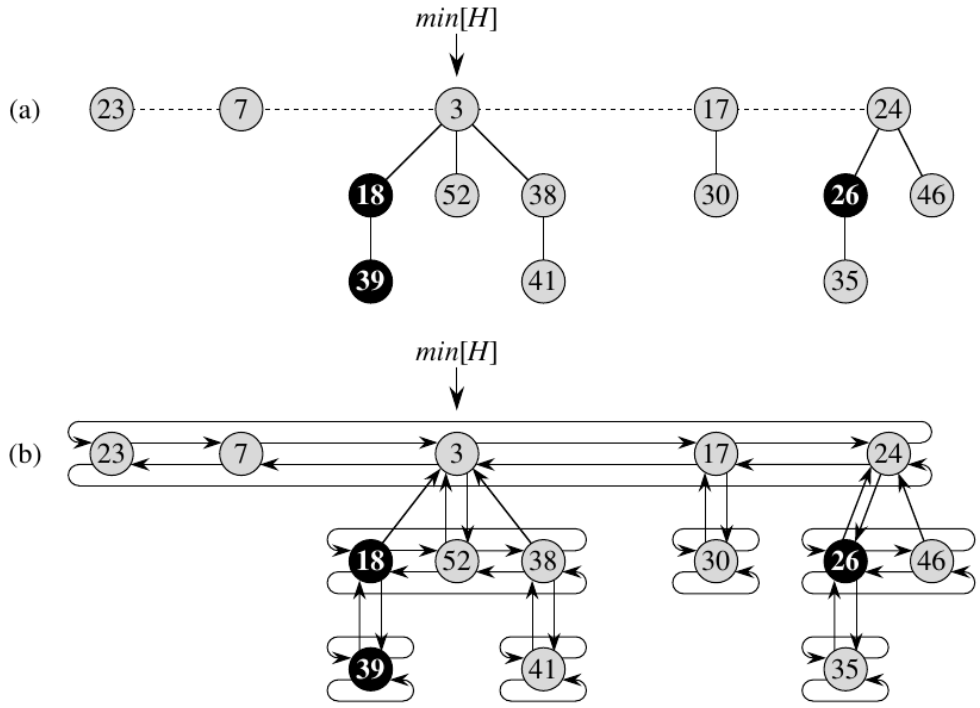
Remark 2 — Circular, doubly linked lists have two advantages for use in Fibonacci heaps.

- We can remove a node from a circular, doubly linked list in $O(1)$ time
- Given two such lists, we can concatenate them (or "splice" them together) into one circular, doubly linked list in $O(1)$ time.

In the descriptions of Fibonacci heap operations, we shall refer to these operations informally.

Two other fields in each node will be of use. The number of children in the ***child list*** of node x is stored in $degree[x]$. The boolean-valued field $mark[x]$ indicates whether node x has lost a child since the last time x was made the child of another node. Newly created nodes are unmarked, and a node x becomes unmarked whenever it is made the child of another node. Until we look at the DECREASE_KEY operation later, we will just set all *mark* fields to FALSE .

A given Fibonacci heap H is accessed by a pointer $min[H]$ to the root of a tree containing a minimum key ; this node is called the ***minimum node*** of the Fibonacci heap. If a Fibonacci heap H is empty, then $min[H]$ = NIL. The roots of all the trees in a Fibonacci heap are linked together using their left and right pointers into a circular, doubly linked list called the ***root list*** of the Fibonacci heap. The pointer $min[H]$ thus points to the node in the root list whose key is minimum. The order of the trees within a root list is arbitrary. We rely on one other attribute for a Fibonacci heap H : the number of nodes currently in H is kept in $n[H]$.



In the above figure (a) is a Fibonacci heap consisting of five min-heap-ordered trees and 14 nodes. The dashed line indicates the root list. The minimum node of the heap is the node containing the key 3. The three marked nodes are blackened and (b) is a more complete representation showing pointers p (up arrows), *child* (down arrows), and *left* and *right* (sideways arrows). These details shall be omitted subsequently, since all the information shown here can be determined from what appears in part (a).

Potential Function

We shall see what is known as the potential method of amortized analysis to analyze the performance of Fibonacci heap operations. For a given Fibonacci heap H, we indicate by $t(H)$ the

number of trees in the root list of H and by $m(H)$ the number of marked nodes in H. The potential of Fibonacci heap H is then defined by

$$\Phi(H) = t(H) + 2m(H)$$

We will gain some intuition for this potential function subsequently. For example, the potential of the Fibonacci heap shown earlier is $5 + 2 \cdot 3 = 11$. The potential of a set of Fibonacci heaps is the sum of the potentials of its constituent Fibonacci heaps. We shall assume that a unit of potential can pay for a constant amount of work, where the constant is sufficiently large to cover the cost of any of the specific constant-time pieces of work that we might encounter.

We assume that a Fibonacci heap application begins with no heaps. The initial potential, therefore, is 0, and by the relation, the potential is non-negative at all subsequent times. An upper bound on the total amortized cost is thus an upper bound on the total actual cost for the sequence of operations.

Maximum degree

The amortized analyses we shall perform subsequently assume that there is a known upper bound $D(n)$ on the maximum degree of any node in an n-node Fibonacci heap. We can that when only the mergeable-heap operations are supported, $D(n) \leq \lfloor \lg n \rfloor$ and when we support DECREASE_KEY and DELETE as well, $D(n) = O(\lg n)$.

Mergeable-heap operations

If only these operations - MAKE_HEAP, INSERT, MINIMUM, EXTRACT_MIN and UNION - are to be supported, each Fibonacci heap is simply a collection of "unordered" binomial trees. An unordered binomial tree is like a binomial tree, and it, too, is defined recursively. The unordered binomial tree U_0 consists of a single node, and an unordered binomial tree U_k consists of two unordered binomial trees U_{k-1} for which the root of one is made into any child of the root of the other. The lemma which gives properties of binomial trees, holds for unordered binomial trees as well, but with the following variation of the last property :

For the unordered binomial tree U_k , the root has degree k, which is greater than that of any other node. The children of the root are roots of subtrees $U_0, U_1, \ldots, U_{k-1}$ in some order.

Thus, if an n-node Fibonacci heap is a collection of unordered binomial trees, then $D(n) = \lg n$. The key idea in the mergeable-heap operations on Fibonacci heaps is to delay work as long as possible. There is a performance trade-off among implementations of the various operations. If the number of trees in a Fibonacci heap is small, then during an EXTRACT_MIN operation we can quickly determine which of the remaining nodes becomes the new minimum node. However, we pay a price for ensuring that the number of trees is small : it can take up to $\Omega(\lg n)$ time to insert a node into a binomial heap or to unite two binomial heaps. As we shall see, we do not attempt to consolidate trees in a Fibonacci heap when we insert a new node or unite two heaps. We save the consolidation for the EXTRACT_M IN operation, which is when we really need to find the new minimum node.

Creating a new Fibonacci heap

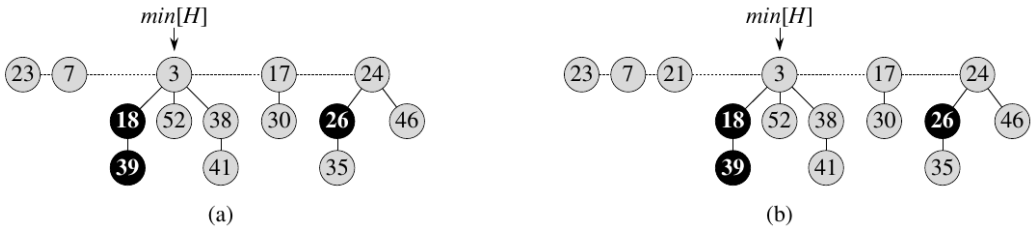
To make an empty Fibonacci heap, the MAKE_FIB_H EAP procedure allocates and returns the Fibonacci heap object H, where $n[H] = 0$ and $min[H] =$ NIL ; there are no trees in H. Since $t(H) = 0$ and $m(H) = 0$, the potential of the empty Fibonacci heap is $\Phi(H) = 0$. The amortized cost of MAKE_FIB_HEAP is thus equal to its $O(1)$ actual cost.

Inserting a node

The following procedure inserts node x into Fibonacci heap H, assuming that the node has already been allocated and that $key[x]$ has already been filled in.

Algorithm FIB_HEAP_INSERT(H, x)

1: $degree[x] \leftarrow 0$
2: $p[x] \leftarrow$ NIL
3: $child[x] \leftrightarrow$ NIL
4: $left[x] \leftrightarrow x$
5: $right[x] \leftrightarrow x$
6: $mark[x] \leftarrow$ FALSE
7: concatenate the root list containing x with root list of H
8: **if** (($min[H] =$ NIL) or ($key[x] < key[min[H]]$))
9: $\quad min[H] \leftarrow x$
10: $n[H] \leftarrow n[H] + 1$



The above figure shows a Fibonacci heap H before (a) and after (b) a node with key 21 has been inserted. After lines 1–6 initialize the structural fields of node x, making it its own circular, doubly linked list, line 7 adds x to the root list of H in $O(1)$ actual time. Thus, node x becomes a single-node min-heap-ordered tree, and thus an unordered binomial tree, in the Fibonacci heap. It has no children and is unmarked. Lines 8–9 then update the pointer to the minimum node of Fibonacci heap H if necessary. Finally, line 10 increments $n[H]$ to reflect the addition of the new node. Unlike the BINOMIAL_HEAP_INSERT procedure, FIB_HEAP_INSERT makes no attempt to consolidate the trees within the Fibonacci heap. If k consecutive FIB_HEAP_INSERT operations occur, then k single-node trees are added to the root list. To determine the amortized cost of FIB_HEAP_INSERT , let H be the input Fibonacci heap and H' be the resulting Fibonacci heap. Then, $t(H') = t(H) + 1$ and $m(H') = m(H)$, and the increase in potential is

$$((t(H) + 1) + 2m(H)) - (t(H) + 2m(H)) = 1$$

Since the actual cost is $O(1)$, the amortized cost is $O(1) + 1 = O(1)$.

Finding the minimum node

The minimum node of a Fibonacci heap H is given by the pointer $min[H]$, so we can find the minimum node in $O(1)$ actual time. Since the potential of H does not change, the amortized cost of this operation is equal to its $O(1)$ actual cost.

Uniting two Fibonacci heaps

The following procedure unites Fibonacci heaps H_1 and H_2, destroying H_1 and H_2 in the process. It simply concatenates the root lists of H_1 and H_2 and then determines the new minimum node.

Algorithm FIB_HEAP_UNION(H_1, H_2)

1: $H \leftarrow$ MAKE_FIB_HEAP()
2: $min[H] \leftarrow min[H_1]$
3: concatenate the root list of H_2 with the root list of H
4: **if** (($min[H_1]$ = NIL) or ($min[H_2] \neq$ NIL and $key[min[H_2]] < key[min[H_1]]$))
5: $\quad min[H] \leftarrow min[H_2]$
6: $n[H] \leftarrow n[H_1] + n[H_2]$
7: free the objects H_1 and H_2
8: **return** H

Lines 1–3 concatenate the root lists of H_1 and H_2 into a new root list H. Lines 2, 4, and 5 set the minimum node of H, and line 6 sets $n[H]$ to the total number of nodes. The Fibonacci heap objects H_1 and H_2 are freed in line 7, and line 8 returns the resulting Fibonacci heap H. As in the FIB_HEAP_INSERT procedure, no consolidation of trees occurs. The change in potential is

$$\Phi(H) - (\Phi(H_1) + \Phi(H_2)) = (t(H) + 2m(H)) - ((t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2))) = 0$$

because $t(H) = t(H_1)+t(H_2)$ and $m(H) = m(H_1)+m(H_2)$. The amortized cost of FIB_HEAP_UNION is therefore equal to its $O(1)$ actual cost.

Extracting the minimum node

The process of extracting the minimum node is the most complicated of the operations presented. It is also where the delayed work of consolidating trees in the root list finally occurs. The following pseudocode extracts the minimum node. The code assumes for convenience that when a node is removed from a linked list, pointers remaining in the list are updated, but pointers in the extracted node are left unchanged. It also uses the auxiliary procedure CONSOLIDATE, which will be presented shortly.

Algorithm FIB_HEAP_EXTRACT_MIN(H)

```
1:  z ← min[H]
2:  if (z ≠ NIL)
3:      for (each child x of z)
4:          add x to the root list of H
5:          p[x] ← NIL
6:      remove z from the root list of H
7:      if (z = right[z])
8:          min[H] ← NIL
9:      else
10:         min[H] ← right[z]
11:         CONSOLIDATE(H)
12:     n[H] ← n[H] − 1
13: return z
```

FIB_HEAP_EXTRACT_MIN works by first making a root out of each of the minimum node's children and removing the minimum node from the root list. It then consolidates the root list by linking roots of equal degree until at most one root remains of each degree. We start in line 1 by saving a pointer z to the minimum node; this pointer is returned at the end. If z = NIL, then Fibonacci heap H is already empty and we are done. Otherwise, we delete node z from H by making all of z's children roots of H in lines 3–5 (putting them into the root list) and removing z from the root list in line 6. If $z = right[z]$ after line 6, then z was the only node on the root list and it had no children, so all that remains is to make the Fibonacci heap empty in line 8 before returning z. Otherwise, we set the pointer $min[H]$ into the root list to point to a node other than z (in this case, $right[z]$), which is not necessarily going to be the new minimum node when FIB_HEAP_EXTRACT_MIN is done.

The next step, in which we reduce the number of trees in the Fibonacci heap, is ***consolidating*** the root list of H ; this is performed by the call CONSOLIDATE(H). Consolidating the root list consists of repeatedly executing the following steps until every root in the root list has a distinct degree value.

(1) Find two roots x and y in the root list with the same degree, where $key[x] \leq key[y]$

(2) ***Link*** y to x : remove y from the root list, and make y a child of x. This operation is performed by the FIB_HEAP_LINK procedure. The field $degree[x]$ is incremented, and the mark on y, if any, is cleared.

The procedure CONSOLIDATE uses an auxiliary array $A[0 \ldots D(n[H])]$; if $A[i] = y$, then y is currently a root with $degree[y] = i$. In detail, the CONSOLIDATE procedure works as follows. Lines 1–2 initialize A by making each entry NIL. The for loop of lines 3–13 processes each root w in the root list. After processing each root w, it ends up in a tree rooted at some node x, which may or may not be identical to w. Of the processed roots, no others will have the same degree as x, and so we will set array entry $A[degree[x]]$ to point to x. When this ***for*** loop terminates, at most one root of each degree will remain, and the array A will point to each remaining root. The

while loop of lines 6–12 repeatedly links the root x of the tree containing node w to another tree whose root has the same degree as x, until no other root has the same degree. This ***while*** loop maintains the following invariant :

At the start of each iteration of the ***while*** loop, $d = degree[x]$.

We use this loop invariant as follows :

Initialization : Line 5 ensures that the loop invariant holds the first time we enter the loop.

Maintenance : In each iteration of the***while*** loop, $A[d]$ points to some root y. Since $d = degree[x] = degree[y]$, we want to link x and y. Whichever of x and y has the smaller key becomes the parent of the other as a result of the link operation, and so lines 8–9 exchange the pointers to x and y if necessary. Next, we link y to x by the call FIB_HEAP_LINK(H, y, x) in line 10. This call increments $degree[x]$ but leaves $degree[y]$ as d. Since node y is no longer a root, the pointer to it in array A is removed in line 11. Because the call of FIB_HEAP_LINK increments the value of $degree[x]$, line 12 restores the invariant that $d = degree[x]$.

Termination : We repeat the while loop until $A[d]$ = NIL, in which case there is no other root with the same degree as x. After the ***while*** loop terminates, we set $A[d]$ to x in line 13 and perform the next iteration of the for loop.

Algorithm CONSOLIDATE(H)

```
1:  for (i ← 0 to D(n[H]))
2:      A[i] ← NIL
3:  for (each node w in the root list of H)
4:      x ← w
5:      d ← degree[x]
6:      while (A[d] ≠ NIL)
7:          y ← A[d]
8:          if (key[x] > key[y])
9:              exchange x ↔ y
10:         FIB_HEAP_LINK(H, y, x)
11:         A[d] ← NIL
12:         d ← d + 1
13:     A[d] ← x
14: min[H] ← NIL
15: for (i ← 0 to D(n[H]))
16:     if (A[i] ≠ NIL)
17:         add A[i] to the root list of H
18:         if ((min[H] = NIL) or (key[A[i]] < key[min[H]]))
19:             min[H] ← A[i]
```

All that remains is to clean up. Once the for loop of lines 3–13 completes, line 14 empties the root list, and lines 15–19 reconstruct it from the array A. After consolidating the root list,

FIB_HEAP_EXTRACT_MIN finishes up by decrementing $n[H]$ in line 12 and returning a pointer to the deleted node z in line 13.

Algorithm FIB_HEAP_LINK(H, y, x)

1: remove y from the root list of H
2: make y a child of x, incrementing $degree[x]$
3: $mark[y] \leftarrow$ FALSE

Observe that if all trees in the Fibonacci heap are unordered binomial trees before FIB_HEAP_EXTRACT_MIN is executed, then they are all unordered binomial trees afterward. There are two ways in which trees are changed. First, in lines 3–5 of FIB_HEAP_EXTRACT_MIN, each child x of root z becomes a root. Each new tree is itself an unordered binomial tree. Second, trees are linked by FIB_HEAP_LINK only if they have the same degree. Since all trees are unordered binomial trees before the link occurs, two trees whose roots each have k children must have the structure of U_k. The resulting tree therefore has the structure of U_{k+1}. We are now ready to show that the amortized cost of extracting the minimum node of an n-node Fibonacci heap is $O(D(n))$. Let H denote the Fibonacci heap just prior to the FIB_HEAP_EXTRACT_MIN operation.

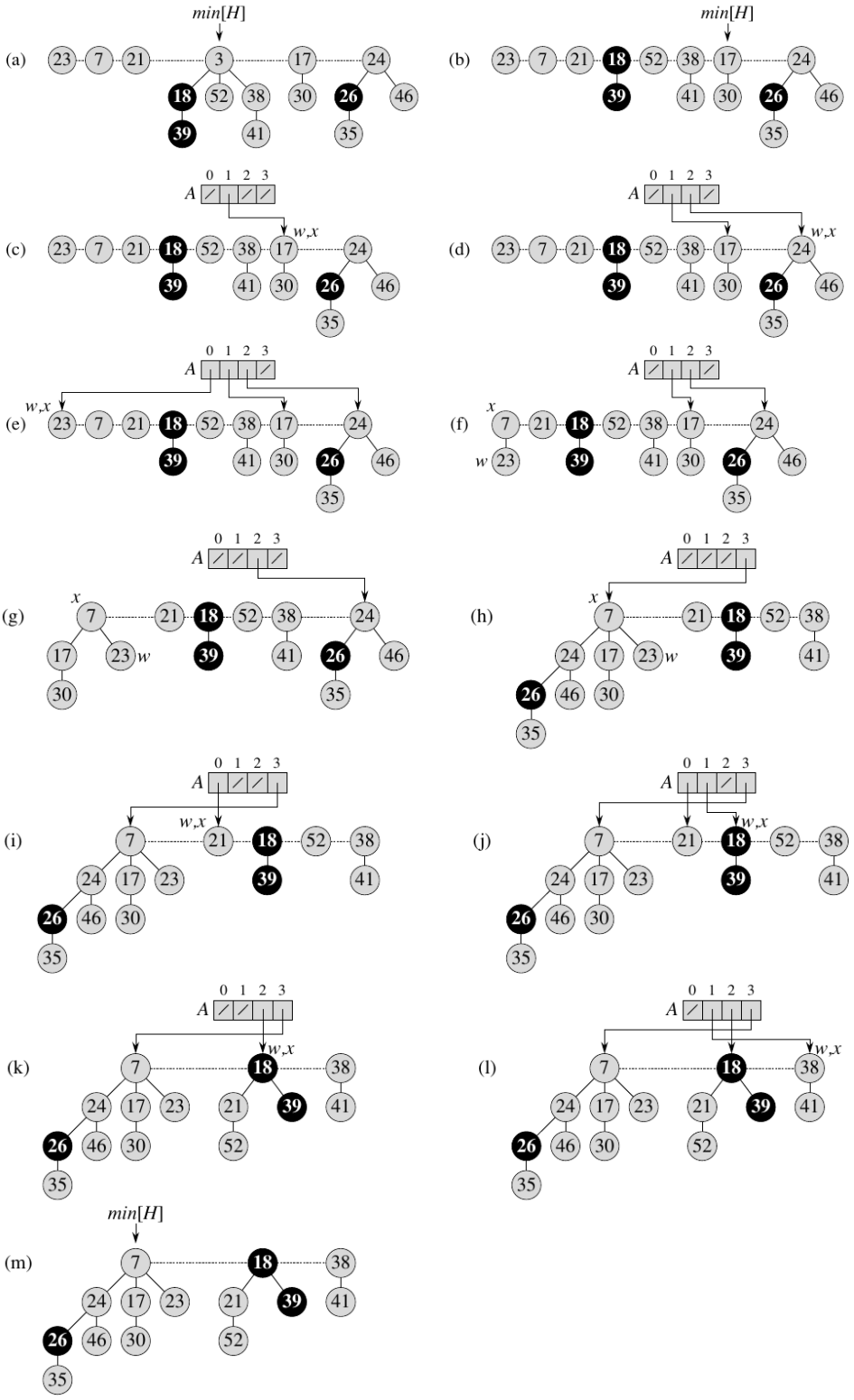
The actual cost of extracting the minimum node can be accounted for as follows. An $O(D(n))$ contribution comes from there being at most $D(n)$ children of the minimum node that are processed in FIB_HEAP_EXTRACT_MIN and from the work in lines 1–2 and 14–19 of CONSOLIDATE. It remains to analyze the contribution from the for loop of lines 3–13. The size of the root list upon calling CONSOLIDATE is at most $D(n) + t(H) - 1$, since it consists of the original $t(H)$ root-list nodes, minus the extracted root node, plus the children of the extracted node, which number at most $D(n)$. Every time through the while loop of lines 6–12, one of the roots is linked to another, and thus the total amount of work performed in the for loop is at most proportional to $D(n)+t(H)$. Thus, the total actual work in extracting the minimum node is $O(D(n)+t(H))$.

The potential before extracting the minimum node is $t(H) + 2m(H)$, and the potential afterward is at most $(D(n) + 1) + 2m(H)$, since at most $D(n) + 1$ roots remain and no nodes become marked during the operation. The amortized cost is thus at most

$$O(D(n) + t(H)) + ((D(n) + 1) + 2m(H)) - (t(H) + 2m(H)) = O(D(n)) + O(t(H)) - t(H) = O(D(n))$$

since we can scale up the units of potential to dominate the constant hidden in $O(t(H))$. Intuitively, the cost of performing each link is paid for by the reduction in potential due to the link's reducing the number of roots by one. We shall see $D(n) = O(\lg n)$, so that the amortized cost of extracting the minimum node is $O(\lg n)$

Below figure shows the action of FIB_HEAP_EXTRACT_MIN . (a) A Fibonacci heap H (b) The situation after the minimum node z is removed from the root list and its children are added to the root list (c)–(e) The array A and the trees after each of the first three iterations of the for loop of lines 3–13 of the procedure CONSOLIDATE. The root list is processed by starting at the node pointed to by $min[H]$ and following right pointers. Each part shows the values of w and x at the end of an iteration. (f)–(h) The next iteration of the for loop, with the values of w and x shown at


(a)
min[H]
23 7 21 3 17 24
18 52 38 30 26 46
39 41 35
(b)
min[H]
23 7 21 18 52 38 17 24
39 41 30 26 46
35
(c)
A
0 1 2 3
w,x
23 7 21 18 52 38 17 24
39 41 30 26 46
35
(d)
A
0 1 2 3
w,x
23 7 21 18 52 38 17 24
39 41 30 26 46
35
(e)
A
0 1 2 3
w,x
23 7 21 18 52 38 17 24
39 41 30 26 46
35
(f)
A
0 1 2 3
x
w
7 21 18 52 38 17 24
23 39 41 30 26 46
35
(g)
A
0 1 2 3
x
7 21 18 52 38 24
17 23 w 39 41 26 46
30 35
(h)
A
0 1 2 3
x
7 21 18 52 38
24 17 23 w 39 41
26 46 30
35
(i)
A
0 1 2 3
w,x
7 21 18 52 38
24 17 23 39 41
26 46 30
35
(j)
A
0 1 2 3
w,x
7 21 18 52 38
24 17 23 39 41
26 46 30
35
(k)
A
0 1 2 3
w,x
7 18 38
24 17 23 21 39 41
26 46 30 52
35
(l)
A
0 1 2 3
w,x
7 18 38
24 17 23 21 39 41
26 46 30 52
35
(m)
min[H]
7 18 38
24 17 23 21 39 41
26 46 30 52
35

the end of each iteration of the while loop of lines 6–12. Part (f) shows the situation after the first time through the while loop. The node with key 23 has been linked to the node with key 7, which is now pointed to by x. In part (g), the node with key 17 has been linked to the node with key 7, which is still pointed to by x. In part (h), the node with key 24 has been linked to the node with key 7. Since no node was previously pointed to by $A[3]$, at the end of the for loop iteration, $A[3]$ is set to point to the root of the resulting tree. (i)–(l) The situation after each of the next four iterations of the for loop. (m) Fibonacci heap H after reconstruction of the root list from the array A and determination of the new $min[H]$ pointer. Now, we shall see how to decrease the key of a node in a Fibonacci heap in $O(1)$ amortized time and how to delete any node from an n-node Fibonacci heap in $O(D(n))$ amortized time. These operations do not preserve the property that all trees in the Fibonacci heap are unordered binomial trees. They are close enough, however, that we can bound the maximum degree $D(n)$ by $O(\lg n)$. We shall proving this bound later, which will imply that FIB_HEAP_EXTRACT_MIN and FIB_HEAP_DELETE run in $O(\lg n)$ amortized time.

Decreasing a key

Algorithm FIB_HEAP_DECREASE_KEY(H, x, k)

```
1: if k > key[x]
2:         return "error : new key is greater than current key"
3: key[x] ← k
4: y ← p[x]
5: if ((y ≠ NIL) and (key[x] < key[y]))
6:         CUT(H, x, y)
7:         CASCADING_CUT(H, y)
8: if (key[x] < key[min[H]])
9:         min[H] ← x
```

Algorithm CUT(H, x, y)

```
1: remove x from the child list of y, decrementing degree[y]
2: add x to the root list of H
3: p[x] ← NIL
4: mark[x] ← FALSE
```

Algorithm CASCADING_CUT(H, y)

```
1: z ← p[y]
2: if z ≠ NIL
3:         if (mark[y] = FALSE)
4:                 mark[y] ← TRUE
5:         else
6:                 CUT(H, y, z)
7:                 CASCADING_CUT(H, z)
```

In the above pseudocode for the operation FIB_HEAP_DECREASE_KEY, we assume as before that removing a node from a linked list does not change any of the structural fields in the removed node. The FIB_HEAP_DECREASE_KEY procedure works as follows. Lines 1–3 ensure that the new key is no greater than the current key of x and then assign the new key to x. If x is a root or if $key[x] \geq key[y]$, where y is x's parent, then no structural changes need occur, since min-heap order has not been violated. Lines 4–5 test for this condition. If min-heap order has been violated, many changes may occur. We start by ***cutting*** x in line 6. The CUT procedure "cuts" the link between x and its parent y, making x a root. We use the mark fields to obtain the desired time bounds. They record a little piece of the history of each node. Suppose that the following events have happened to node x :

1. at some time, x was a root
2. then x was linked to another node
3. then two children of x were removed by cuts

As soon as the second child has been lost, we cut x from its parent, making it a new root. The field $mark[x]$ is TRUE if steps 1 and 2 have occurred and one child of x has been cut. The CUT procedure, therefore, clears $mark[x]$ in line 4, since it performs step 1. (We can now see why line 3 of FIB_HEAP_LINK clears $mark[y]$: node y is being linked to another node, and so step 2 is being performed. The next time a child of y is cut, $mark[y]$ will be set to TRUE).

We are not yet done, because x might be the second child cut from its parent y since the time that y was linked to another node. Therefore, line 7 of FIB_HEAP_DECREASE_KEY performs a ***cascading-cut*** operation on y. If y is a root, then the test in line 2 of CASCADING_CUT causes the procedure to just return. If y is unmarked, the procedure marks it in line 4, since its first child has just been cut, and returns. If y is marked, however, it has just lost its second child; y is cut in line 6, and CASCADING_CUT calls itself recursively in line 7 on y's parent z. The CASCADING_CUT procedure recurses its way up the tree until either a root or an unmarked node is found.

Once all the cascading cuts have occurred, lines 8–9 of FIB_HEAP_DECREASE_KEY finish up by updating $min[H]$ if necessary. The only node whose key changed was the node x whose key decreased. Thus, the new minimum node is either the original minimum node or node x.

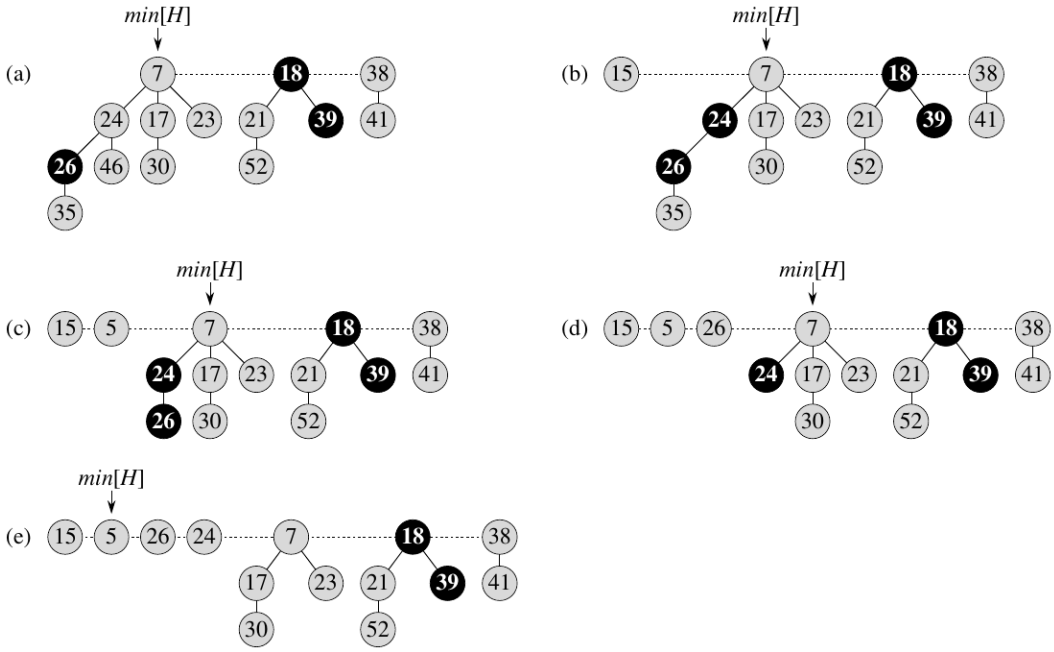
We shall now show that the amortized cost of FIB_HEAP_DECREASE_KEY is only $O(1)$. We start by determining its actual cost. The FIB_HEAP_DECREASE_KEY procedure takes $O(1)$ time, plus the time to perform the cascading cuts. Suppose that CASCADING_CUT is recursively called c times from a given invocation of FIB_HEAP_DECREASE_KEY. Each call of CASCADING_CUT takes $O(1)$ time exclusive of recursive calls. Thus, the actual cost of FIB_HEAP_DECREASE_KEY, including all recursive calls, is $O(c)$.

We next compute the change in potential. Let H denote the Fibonacci heap just prior to the FIB_HEAP_DECREASE_KEY operation. Each recursive call of CASCADING_CUT, except

for the last one, cuts a marked node and clears the mark bit. Afterward, there are $t(H) + c$ trees (the original $t(H)$ trees, $c - 1$ trees produced by cascading cuts, and the tree rooted at x) and at most $m(H) - c + 2$ marked nodes ($c - 1$ were unmarked by cascading cuts and the last call of CASCADING_CUT may have marked a node). The change in potential is therefore at most

$$((t(H) + c) + 2(m(H) - c + 2)) - (t(H) + 2m(H)) = 4 - c$$

Thus, the amortized cost of FIB_HEAP_DECREASE_KEY is at most $O(c) + 4 - c = O(1)$, since we can scale up the units of potential to dominate the constant hidden in $O(c)$. We can now see why the potential function was defined to include a term that is twice the number of marked nodes. When a marked node y is cut by a cascading cut, its mark bit is cleared, so the potential is reduced by 2. One unit of potential pays for the cut and the clearing of the mark bit, and the other unit compensates for the unit increase in potential due to node y becoming a root.



The above figure shows two calls of FIB_HEAP_DECREASE_KEY (a) The initial Fibonacci heap. (b) The node with key 46 has its key decreased to 15. The node becomes a root, and its parent (with key 24), which had previously been unmarked, becomes marked. (c)–(e) The node with key 35 has its key decreased to 5. In part (c), the node, now with key 5, becomes a root. Its parent, with key 26, is marked, so a cascading cut occurs. The node with key 26 is cut from its parent and made an unmarked root in (d). Another cascading cut occurs, since the node with key 24 is marked as well. This node is cut from its parent and made an unmarked root in part (e). The cascading cuts stop at this point, since the node with key 7 is a root. (Even if this node were not a root, the cascading cuts would stop, since it is unmarked.) The result of the FIB_HEAP_DECREASE_KEY operation is shown in part (e), with $min[H]$ pointing to the new minimum node.

Deleting a node

It is easy to delete a node from an n-node Fibonacci heap in $O(D(n))$ amortized time, as is done by the following pseudocode. We assume that there is no key value of $-\infty$ currently in the Fibonacci heap.

Algorithm FIB_HEAP_DELETE(H, x)

1: FIB_HEAP_DECREASE_KEY($H, x, -\infty$)
2: FIB_HEAP_EXTRACT_MIN(H)

FIB_HEAP_DELETE is analogous to BINOMIAL_HEAP_DELETE . It makes x become the minimum node in the Fibonacci heap by giving it a uniquely small key of $-$ inf. Node x is then removed from the Fibonacci heap by the FIB_HEAP_ EXTRACT_MIN procedure. The amortized time of FIB_HEAP_DELETE is the sum of the $O(1)$ amortized time of FIB_HEAP_DECREASE_ KEY and the $O(D(n))$ amortized time of FIB_HEAP_EXTRACT_MIN. Since we shall see subsequently, that $D(n) = O(\lg n)$, the amortized time of FIB_HEAP_DELETE is $O(\lg n)$.

Bounding the maximum degree

To prove that the amortized time of FIB_HEAP_EXTRACT_MIN and FIB_HEAP_DELETE is $O(\lg n)$, we must show that the upper bound $D(n)$ on the degree of any node of an n-node Fibonacci heap is $O(\lg n)$. When all trees in the Fibonacci heap are unordered binomial trees, $D(n) = \lfloor \lg n \rfloor$. The cuts that occur in FIB_HEAP_DECREASE_KEY, however, may cause trees within the Fibonacci heap to violate the unordered binomial tree properties. In this section, we shall show that because we cut a node from its parent as soon as it loses two children, $D(n)$ is $O(\lg n)$. In particular, we shall show that $D(n) \leq \lfloor \log_\phi n \rfloor$, where $\phi = (1 + \sqrt{5})/2$. The key to the analysis is as follows. For each node x within a Fibonacci heap, define $size(x)$ to be the number of nodes, including x itself, in the subtree rooted at x. (Note that x need not be in the root list—it can be any node at all.) We shall show that $size(x)$ is exponential in $degree[x]$. Bear in mind that $degree[x]$ is always maintained as an accurate count of the degree of x.

Lemma 2

Let x be any node in a Fibonacci heap, and suppose that $degree[x] = k$. Let $y_1, y_2, \ldots, y_k$ denote the children of x in the order in which they were linked to x, from earliest to the latest. The, $degree[y_1] \geq 0$ and $degree[y_i] \geq i - 2$ for $i = 2, 3, \ldots, k$.

Proof : Obviously $degree[y_1] \geq 0$. For $i \geq 2$, we note that when y_i was linked to x, all of $y_1, y_2, \ldots, y_{k-1}$ were children of x, so we must have had $degree[x] = i - 1$. Node y_i is linked to x only if $degree[x] = degree[y_i]$, so we must have also had $degree[y_i] = i - 1$ at that time. Since then, node y_i has lost at most one child, since it would have been cut from x if it had lost two children. We conclude that $degree[y_i] \geq i - 2$. ■

We finally come to the part of the analysis that explains the name "Fibonacci heaps." Note

that the k-th Fibonacci number is defined by the recurrence

$$F_k = \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{if } k = 1 \\ F_{k-1} + F_{k-2} & \text{if } k \geq 2 \end{cases}$$

The following lemma gives another way to express F_k.

Lemma 3

For all integers $k \geq 0$,

$$F_{k+2} = 1 + \sum_{i=0}^{k} F_i$$

Proof : The proof is by induction on k. When $k = 0$,

$$1 + \sum_{i=0}^{0} F_i = 1 + F_0 \ = 1 + 0 = 1 = F_2$$

We now assume the inductive hypothesis that $F_{k+1} = 1 + \sum_{i=0}^{k-1} F_i$, and we have $F_{k+2} = F_k + F_{k+1} = F_k + \left(1 + \sum_{i=0}^{k-1} F_i\right) = 1 + \sum_{i=0}^{k} F_i$ ■

The following lemma and its corollary complete the analysis. They use the inequality $F_{k+2} \geq \phi^k$ where ϕ as defined earlier is the golden ratio.

Lemma 4

Let x be any node in a Fibonacci heap, and let $k = degree[x]$. Then, $size(x) \geq F_{k+2} \geq \phi^k$, where $\phi = (1 + \sqrt{5})/2$.

Proof : Let s_k denote the minimum possible value of $size(z)$ over all nodes z such that $degree[z] = k$. Trivially, $s_0 = 1, s_1 = 2$ and $s_2 = 3$. The number s_k is at most $size(x)$ and clearly, the value of s_k increases monotonically with k. Let $y_1, y_2, \ldots, y_k$ denote the children of x in the order in which they were linked to x. To compute a lower bound on $size(x)$, we count one for x itself and one for the first child y_1 (for which $size(y_1)) \geq 1$ giving

$$size(x) \geq s_k = 2 + \sum_{i=2}^{k} s_{degree[y_i]} \geq 2 + \sum_{i=2}^{k} s_{i-2}$$

where the last inequality follows from the earlier result that $degree[y_i] \geq i-2$ and the monotonicity of s_k so that $s_{degree[y_i]} \geq s_{i-2}$. We now show by induction on k that $s_k \geq F_{k+2}$ for all non-negative integer k. The bases, for $k = 0$ and $k = 1$ are trivial. For the inductive step, we assume that $k \geq 2$ and that $s_i \geq F_{i+2}$ for $i = 0, 1, \ldots, k-1$. We have

$$s_k \geq 2 + \sum_{i=2}^{k} s_{i-2} \geq 2 + \sum_{i=2}^{k} F_i = 1 + \sum_{i=0}^{k} F_i = F_{k+2}$$

Thus, we have shown that $size(x) \geq s_k \geq F_{k+2} \geq \phi^k$ ■

Now, let x be any node in an n-node Fibonacci heap and let $k = degree[x]$. We now know that $n \geq size(x) \geq \phi^k$. Taking base-$\phi$ logarithms gives us $k \leq \log_\phi n$. In fact, because k is an integer, $k \leq \lfloor \log_\phi n \rfloor$. The maximum degree $D(n)$ of any node is thus $O(\lg n)$.

III

Algorithms

7 Greedy Algorithms

§7.1 Introduction

Greedy algorithms constitute an important class of problems and as we shall see, that if the underlying mathematical structure is a *matroid*, then it always works. A natural class of problems which arise are optimisation problems. For example : in the shortest path plan from A to B, with each edge having a cost function c, one example of a greedy approach would be to select the path having the least weight at each node. However, we can clearly see that this need not always be the optimal solution i.e., the shortest path. We shall now look at some generic greedy algorithms with proof strategies starting with the scheduling problem also referred to as Activity / Interval selection problem. Before that, we shall see the following mathematical structure called ***matroids.***

§7.2 Matroids

A ***matroid*** is an ordered pair $M = (S, I)$ satisfying the following conditions

a. S is a finite set

b. I is a nonempty family of subsets of S ($|I| \leq 2^{|S|}$), such that if $B \in I$ and $A \subseteq B$, then $A \in I$. We say that I is ***hereditary*** if it satisfies this property. Note that the empty set ϕ is necessarily a member of I

c. If $A \in I, B \in I$ and $|A| < |B|$, then there exist some element $x \in B - A$ such that $A \cup \{x\} \in I$. We say M satisfies the ***exchange*** property.

The sets in I are typically called independent sets. The union of all sets in I is called ground set. An independent set is called basis if it is not a proper subset of another independent set. Now suppose each element of the ground set of a matroid M is given an arbitrary non-negative weight. The matroid optimization problem is to compute a basis with maximum total weight. The following natural greedy strategy computes a basis for any weighted matroid. We will prove that for any matroid M and any weight function w, GREEDYBASIS(M, w) returns A, a maximum weight basis of M ($w(A) = \sum_{a \in A} w(a)$). We assume that if $S = \{s_1, s_2 \ldots, s_n\}$, then $w(s_1) \geq w(s_2) \geq \ldots w(s_n)$

Algorithm GREEDYBASIS(M, w)

```
1: A ← φ
2: for i = 1 to n
3:         if (A + s_i ∈ I)
4:                 A = A + s_i
5: return A
```

We now prove that algorithm does indeed return the maximum weight basis of M. Let $A = \{a_1, \ldots, a_n\}$ be returned by the algorithm. Now suppose there exists an optimal solution $O \in I$, $O = \{o_1, \ldots, o_t\}$. We present a two step proof wherein first we prove that the cardinality of the sets are equal and then prove that given the same cardinality, they have the same weight.

If $|A| > |O|$, then by the exchange property $\exists\, y \in A - O$, such that $O + y \in I$. However as per assumption $w(O + y) > w(O)$, which contradicts the fact that O is an optimal solution. Thus, $n \leq t$. If $|A| < |O|$, then again by exchange property $\exists\, y \in O - A$, such that $A + y \in I$. However this is a contradiction because if any other element could be added to A to obtain a larger set, the greedy algorithm would have added it. Thus, $n \geq t$. Combining both the inequalities, we get $n = t = k$ (say).

Thus, now $A = \{a_1, \ldots, a_k\}$ and $O = \{o_1, \ldots, o_k\}$ such that $w(a_1) \geq w(a_2) \geq \ldots w(a_k)$ and $w(o_1) \geq w(o_2) \geq \ldots w(o_k)$. If A were not to be the optimal solution then $w(A) \leq w(O)$. We will prove by contradiction that $w(A) \not< w(O) \rightarrow w(A) = w(O)$. Assume

$$\sum_{i=1}^{k} w(a_i) < \sum_{i=1}^{k} w(o_i)$$

Then $\exists\, t$ such that $w(a_t) < w(o_t)$. (In case of multiple such indices, we take the smallest index). We now consider the following two sets

$$A^{'} = \{a_1, \ldots, a_{t-1}\} \in I$$

$$O^{'} = \{o_1, \ldots, o_t\} \in I$$

Then $\exists\, o_s \in O^{'}$ such that $A^{'} \cup o_s \in I$. However, $w(o_s) \geq w(o_t) > w(a_t)$. Thus, the greedy algorithm considers and rejects the heavier element o_s before it considers the lighter element a_t however this is a contradiction because the greedy algorithm accepts elements in decreasing order of weight. Hence proved.

§7.3 Scheduling Problem 1

We have a set $\mathbb{A} = \{A_1, \ldots, A_n\}$ of intervals. For $1 \leq i \leq n$, the interval A_i has the form $A_i = [s_i, f_i)$ where s_i is the start time of interval A_i and f_i is the finish time of A_i. The $s_i^{'}$s and $f_i^{'}$s are assumed to be positive integers, and we assume that $s_i < f_i$ for all i. An optimal solution would be a subset $\mathbb{B} \subseteq \mathbb{A}$ of **pairwise disjoint intervals** of maximum size (i.e., one that maximizes $|\mathbb{B}|$).

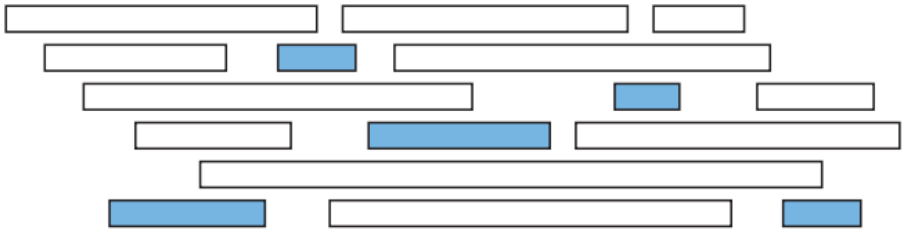


Fig. A maximal conflict-free schedule for a set of intervals

One of the greedy strategies would be based on minimum duration. Strategy is to sort the intervals in increasing order of duration. At any stage, choose the interval of minimum duration that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is the value $f_i - s_i$). To show that a greedy algorithm is not correct, it suffices to provide a single counterexample. For this strategy, we see that if we have $\mathbb{A} = \{[0, 5), [6, 10), [4, 7)\}$, then the algorithm chooses $[4, 7)$ and we're done, however the other two intervals comprise an optimal solution.

Another greedy approach would be to choose the interval with minimum number of overlapping intervals i.e. one with fewest conflicts. For each job A_i, count the number of conflicting jobs c_i and then schedule in ascending order of c_i.

counterexample for shortest interval

counterexample for fewest conflicts

Thus, as we see several greedy strategies can be proposed. However, the one we will propose now is based on the ***earliest finish time first*** approach, which we will prove is optimal using a strategy called ***greedy stays ahead***. In this strategy, we sort the intervals in increasing order of finishing times and then at any stage, we choose the earliest finishing interval that is disjoint from all previously chosen intervals (i.e., the local evaluation criterion is the value f_i). We argue that the greedy solution returned is the optimal solution. Let $\mathbb{B}$ be the greedy solution, then

$$\mathbb{B} = (A_{i_1}, \ldots, A_{i_k})$$

where $i_1 < \cdots < i_k$. Let $\mathbb{O}$ be any optimal solution, then

$$\mathbb{O} = (A_{j_1}, \ldots, A_{j_l})$$

where $j_1 < \cdots < j_l$. Now, since $\mathbb{O}$ is optimal, $l \geq k$. We need to prove that $l = k$. We will assume $l > k$ and obtain a contradiction. The correctness proof depends on the greedy stays ahead lemma. This phrase indicates that the greedy solution is in fact the "best possible" partial solution at every stage of the algorithm.

Lemma *(Greedy Stays Ahead)*: For all $m \geq 1$, it holds that

$$f_{i_m} \leq f_{j_m}$$

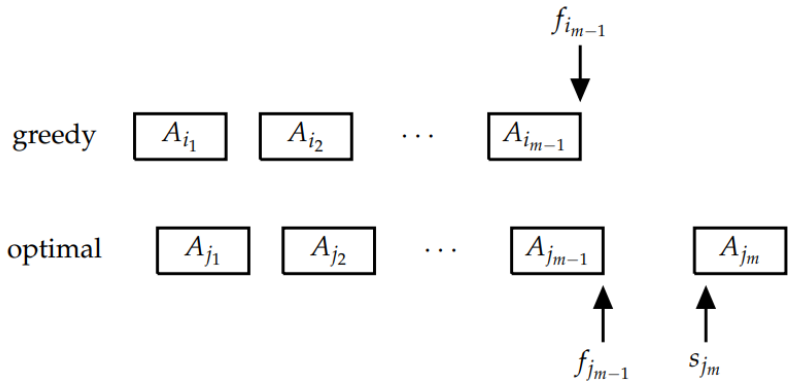
Proof : The initial case is $m = 1$. We have $f_{i_1} \leq f_{j_1}$ because the greedy algorithm begins by choosing $i_1 = 1$ (A_1 has the earliest finishing time). Our induction assumption is that

$$f_{i_{m-1}} \leq f_{j_{m-1}}$$

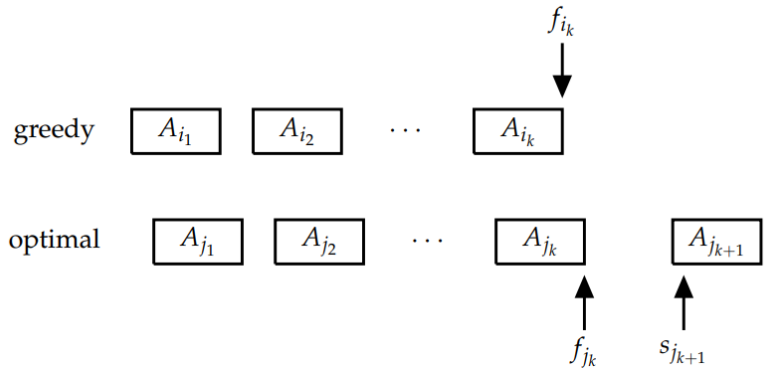
Now consider A_{i_m} and A_{j_m}. We have

$$s_{j_m} \geq f_{j_{m-1}} \geq f_{i_{m-1}}$$

A_{i_m} has the earliest finishing time of any interval that starts after $f_{i_{m-1}}$ finishes. Therefore, $f_{i_m} \leq f_{j_m}$ ■



Thus, from the lemma we have $f_{i_k} \leq f_{j_k}$. Now, suppose $l > k$. $A_{j_{k+1}}$ starts after A_{j_k} finishes, and $f_{i_k} \leq f_{j_k}$. So, $A_{j_{k+1}}$ is a feasible interval with respect to the greedy algorithm, and therefore the greedy solution would not have terminated with A_{i_k}. This contradiction shows that $l = k$.



Another proof technique used for Greedy algorithms involve ***exchange arguments*** wherein we compare a greedy solution $\mathbb{G}$ to an optimal solution $\mathbb{O}$, both defined explicitly. We next show that if $\mathbb{G} \neq \mathbb{O}$ then they must differ in some way and define this point of departure to use in the rest of the proof. Then we show how to transform $\mathbb{O}$ to $\mathbb{G}$ by some technique and prove that in by doing so, we did not increase/decrease the cost of $\mathbb{G}$. Finally, we argue that we have decreased the number of differences between $\mathbb{G}$ and $\mathbb{O}$ by performing the exchange, and that by iterating this exchange we can turn $\mathbb{O}$ into $\mathbb{G}$ without impacting the quality of the solution. Therefore, $\mathbb{G}$ must be optimal.

§7.4 Scheduling Problem 2

In this problem we have a set of events denoted by $T_1, T_2, \ldots, T_n$. Every such event T_i is associated with two attributes, namely the length l_i which denotes the length of time interval T_i requires and a deadline d_i by which T_i should be completed. We define schedule to be a permutation of $T_1, T_2, \ldots, T_n$, wherein the events follow one another successively without any time gap between them and denote it as $T_{p_1}, T_{p_2}, \ldots, T_{p_n}$. Thus, the finish time of T_{p_i} denoted as f_i can be obtained by the summation $\sum_{k=1}^{i} l_{p_i}$. Since the events are scheduled without any delay, the start time s_i of T_{p_i} is same as the finish time of $T_{p_{i-1}}$ i.e., f_{i-1} i.e.,

$$f_i(T_{p_i}) = \sum_{k=1}^{i} l_{p_i}$$

$$s_i(T_{p_i}) = f_{i-1}(T_{p_{i-1}})$$

f_i and f_{i-1} are related as follows $f_i = f_{i-1} + l_{p_i}$ with $f_0 = 0$. Thus each interval $T_i(l_i, d_i)$ in a schedule S is associated with a start and finish time (s_i, f_i).

We now define the delay for each interval T_i in a schedule S as follows :

$$D_S(T_i) = \begin{cases} 0 & \text{if } f_i \leq d_i \\ f_i - d_i & \text{if } f_i > d_i \end{cases} \tag{7.1}$$

Delay of a schedule, D for a particular permutation p is then defined as $D_p = \max(D_p(T_1), \ldots, D_p(T_n))$. Since, every schedule has a finite number of events and maximum is well defined over a finite number of elements, every schedule has a delay (which could be zero as well) which can be computed. We have to find a schedule with minimum delay D_p over the set of all permutations.

The greedy strategy used here is ***earliest deadline first***. Thus, the output of the greedy algorithm is a schedule of the form $T_1, T_2, \ldots, T_n$ such that $d_1 \leq d_2 \cdots \leq d_n$. Thus, any other solution will be a permutation of the above schedule. If we define inversion as an adjacent pair of events (T_i, T_j) such that $i < j$ for which the deadlines are out of order i.e., $d_i \geq d_j$, then we can state that the greedy solution has no inversions and any other optimal solution has atleast one inversion. We now use a proof by exchange arguments method to show that any optimal solution can be converted to the greedy solution using a series of cost preserving transformations.

Consider an optimal schedule S_1 which has atleast one inversion pair (T_i, T_j) for which $i < j$ and $d_i \geq d_j$. Now consider a schedule S_2 obtained by interchanging their positions i.e., now the events appear in the order (T_j, T_i). We will now show that two schedules S_1 and S_2 both have the same delay i.e., $D(S_1) = D(S_2)$.

We denote by x and y the delays of events T_j and T_i in S_1 respectively. Similarly the delays of events T_j and T_i in S_2 are denoted by x' and y' respectively i.e., the ordered pairs

$(x, y) = (D_{S_1}(T_j), D_{S_1}(T_i))$ and $(x^{'}, y^{'}) = (D_{S_2}(T_j), D_{S_2}(T_i))$. We observe that $y^{'} \leq y$ and $x^{'} \geq x$. Now, consider $x^{'}$ as shown

$$x^{'} = f_{S_2}(T_j) - d_j = f_{S_1}(T_i) - d_j \leq f_{S_1}(T_i) - d_i = y$$

Thus, we have the following set of inequalities

$$x^{'} \leq y \leq D(S_1)$$

$$y^{'} \leq y \leq D(S_1)$$

We thus conclude that this transformation doesn't increase the maximum i.e., $D(S_2) \leq D(S_1)$. However, since S_1 is an optimal schedule, we have $D(S_1) \leq D(S_2)$. Combining the two inequalities yields $D(S_1) = D(S_2)$. Hence, we have shown that this is a cost-preserving transformation. Thus, given an optimal solution which has an inversion, we tinker it and remove using a finite number of the aforementioned transformations. In this way, theoretically, using a series of exchange arguments, we can convert any optimal solution into the greedy solution.

8 Huffman Codes

§8.1 Introduction

Algorithmic genetic research in the field of computational geometry requires encoding and decoding of several hundreds and thousands of genome sequences. It becomes necessary to develop a binary character code for a given character set which enables us to compress data more effectively. When at the time (1951), this open problem was offered as an escape from the MIT final exam, a Ph.D student named Huffman under Robert Fano came up with an elegant coding scheme just before the day of exam and built a simplistic mathematical proof from the very papers he threw in the trash.

§8.2 Problem

5 (Huffman Coding Scheme)

Character file	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed length code word	000	001	010	011	100	101
Variable-length code word	0	101	100	111	1101	1100

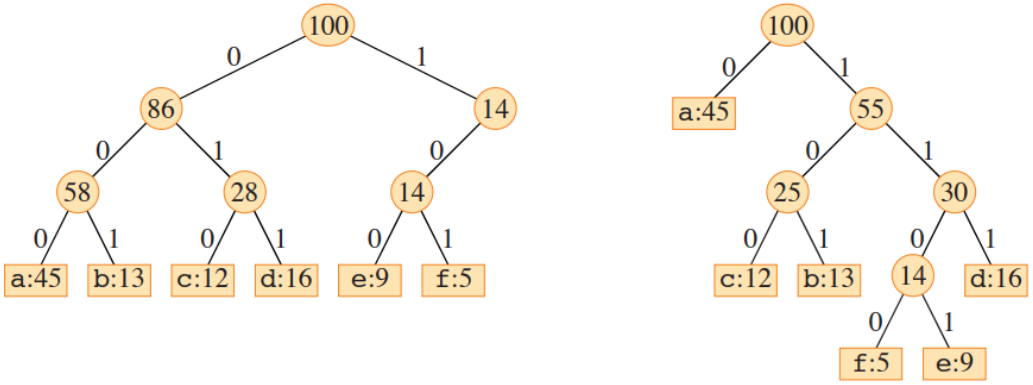
Suppose we have the above character-coding problem : a data file of 100,000 characters contains only the characters $a - f$, with the frequencies indicated in the figure. If we use a fixed-length code, we need 3 bits to represent 6 characters: $a = 000$, $b = 001$, $\ldots$, $f = 101$. This method requires 300,000 bits to code the entire file. Can we do better? A variable-length code can do considerably better than a fixed-length code, by giving frequent characters short codewords and infrequent characters long codewords. The figure shows such a code; here the 1-bit string 0 represents a, and the 4-bit string 1100 represents f. This code requires 224,000 bits to represent the file, a savings of approximately 2%. In fact, this is an optimal character code for this file, as we shall see.

$$(45.1 + 13.3 + 12.3 + 16.3 + 9.4 + 5.4).1,000 = 224,000 \text{ bits}$$

§8.3 Solution

Lets take a small example where in we encode a as 01, b as 11 and c as 0111. Then, while parsing a string 0111, the problem arises as to whether it should be decoded as ab or c. If we notice, the root cause of the problem occurs because the code for a is the codeword for a is a prefix in the codeword for c. The solution is to come up with prefix-free codes or ***prefix codes***. We now discuss a little about it in detail.

Prefix-codes : As mentioned, these are codes in which no code word is also a prefix of some other code word. Any prefix-free binary code can be visualized as a binary tree with the encoded characters stored at the leaves. The code word for any symbol is given by the path from the root to the corresponding leaf where 0 means "go to the left child" and 1 means "go to the right child.". Thus, the length of any symbol's codeword is the depth of the corresponding leaf in the code tree. Note that these are not binary search trees, since the leaves need not appear in sorted order and internal nodes do not contain character keys.



In the figure, (a) represents the tree corresponding to the fixed-length code $a = 000, \ldots f = 101$ and (b) represents the tree corresponding to the optimal prefix code $a = 0, b = 101, \ldots, f = 1100$.

An optimal code for a file is always represented by a full binary tree, in which every non-leaf (internal) node has two children (degree= 2). The fixed-length code in our example is not optimal since its tree, shown in (a), is not a full binary tree because it contains codewords beginning with 10, but none beginning with 11. Since we can now restrict our attention to full binary trees, we can say that if A is the alphabet from which the characters are drawn and all character frequencies are positive, then the tree for an optimal prefix-free code has exactly $|A|$ leaves, one for each letter of the alphabet, and exactly $|A| - 1$ internal nodes.

Given a tree T corresponding to a prefix-free code, we can compute the number of bits required to encode a file. For each character c in the alphabet A, let $f(c)$ denote the frequency of c in the file and let $d_T(c)$ denote the depth of c's leaf in the tree. Note that $d_T(c)$ is also the length of the codeword for character c. The number of bits required to encode a file is given as shown which we call the cost of a tree T.

$$C(T) = \sum_{c \in A} f(c).d_T(c)$$

Thus, the problem now becomes to find an optimal full binary tree with the minimum cost. Huffman developed the following greedy algorithm to produce such an optimal code : ***Merge the***

two least frequent letters and recurse. Given the simple structure of Huffman's algorithm, it's rather surprising that it produces an optimal prefix-free binary code. Fortunately, the recursive structure makes this claim easy to prove using an exchange argument. We start by proving that the algorithm's very first choice is correct.

Lemma ***(Optimal prefix-free codes have the greedy-choice property)*** **:** Let x and y be the two least frequent characters (breaking ties between equally frequent characters arbitrarily). There is an optimal prefix-free code tree in which x and y are siblings and have the largest depth of any leaf.

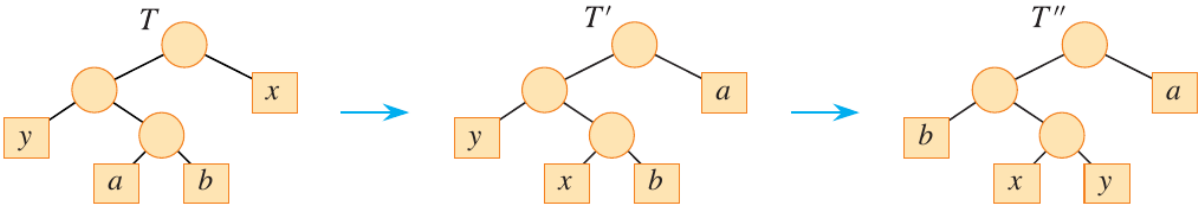
Proof : Let us assume that T is an optimal tree in which x and y, the lowest frequency characters do not occur as siblings at the maximum depth. Say a and b are the two characters present at the maximum depth in T such that $f(a), f(b) > f(x), f(y)$. Now, consider another tree $T^{'}$ in which x and a are swapped. Since cost of a tree is a linear function, we compare the costs of T and $T^{'}$ by computing $C(T) - C(T^{'})$ which is given as $C(T) - C(T^{'})$

$$= d_T(x).f(x) + d_T(a).f(a) - d_{T'}(x).f(x) - d_{T'}(a).f(a)$$

$$= d_T(x).f(x) + d_T(a).f(a) - d_T(a).f(x) - d_T(x).f(a)$$

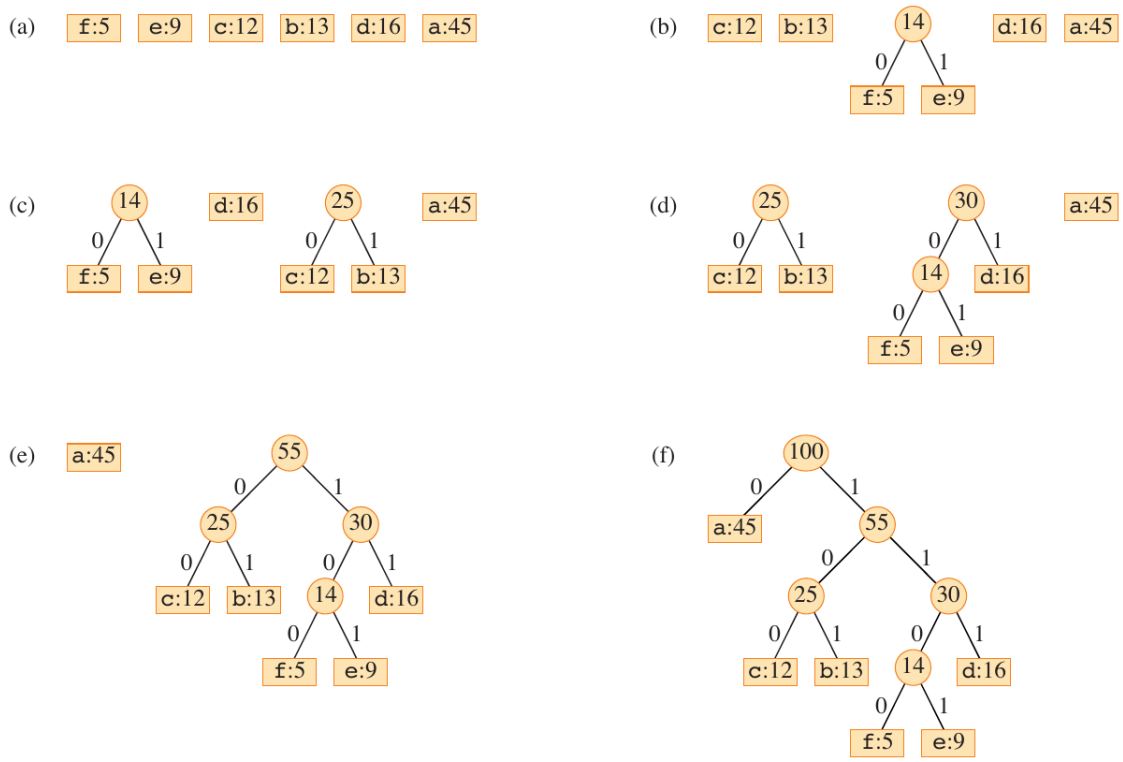
$$= f(x).[d_T(x) - d_T(a)] + f(a).[d_T(a) - d_T(x)]$$

$$= [d_T(a) - d_T(x)].[f(a) - f(x)]$$



It can be seen that both terms involved in the multiplication are positive, hence $C(T) - C(T^{'}) \geq 0 \Rightarrow C(T) \geq C(T^{'})$. However, T being an optimal tree $C(T) \leq C(T^{'})$. Combining the two inequalities yields $C(T) = C(T^{'})$. Similarly, the tree $T^{''}$ obtained by swapping y and b doesn't increase the cost of T. The recursive argument relies on the following non-standard recursive definition : A full binary tree is either a single node, or a full binary tree where some leaf has been replaced by an internal node with two leaf children. The following figure shows each step of the Huffman algorithm worked out on our example.

Each step merges the two trees with the lowest frequencies. Leaves are shown as rectangles containing a character and its frequency. Internal nodes are shown as circles containing the sum of the frequencies of their children. An edge connecting an internal node with its children is labeled 0 if it is an edge to a left child and 1 if it is an edge to a right child. The codeword for a letter is the sequence of labels on the edges connecting the root to the leaf for that letter.



Lemma 5 (*Optimal prefix-free codes have the optimal-substructure property*)

Let x and y be two characters in A with minimum frequency. Let $A^{'}$ be character set A with the characters x and y removed and a new character z added, so that $A^{'} = (A - \{x, y\}) \cup \{z\}$.Define frequency for all characters in $A^{'}$ with the same values as in A , along with $f(z) = f(x) + f(y)$. Let $T^{'}$ be any tree representing an optimal prefix-free code for $A^{'}$. Then the tree T , obtained from $T^{'}$ by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix-free code for the A .

Proof : For each character $c \in A - \{x, y\}$, we have $d_T(c) = d_{T'}(c)$ and hence $f(c).d_T(c) = f(c).d_{T'}(c)$. For x and y, since $d_T(x) = d_T(y) = d_{T'}(z) + 1$, we have $f(x).d_T(x) + f(y).d_T(y) = (f(x) + f(y)).(d_{T'}(z) + 1) = f(z).d_{T'}(z) + (f(x) + f(y))$. Thus, $C(T) = C(T^{'}) + f(x) + f(y)$.

We now prove the lemma by contradiction. Suppose T doesn't represent the optimal prefix-free code for A, then there exits a tree $T^{''}$ such that $C(T^{''}) < C(T)$. WLOG, by previous lemma, $T^{''}$ has x and y as siblings. Let $T^{'''}$ be the tree $T^{''}$ with common parent of X and y replaced by a leaf z such that $f(z) = f(x) + f(y)$. Then $C(T^{'''}) = C(T^{''}) - f(x) - f(z) < C(T) - f(x) - f(z) = C(T^{'})$, yielding a contradiction to the assumption that $T^{'}$ represents the optimal prefix code for $A^{'}$. Thus, T must represent the optimal prefix code for A.

9 Minimum Spanning Tree

§9.1 Introduction

Electronic circuit designs often need to make the pins of several components electrically equivalent by wiring them together. To interconnect a set of n pins, the designer can use an arrangement of $n - 1$ wires, each connecting two pins. Of all such arrangements, the one that uses the least amount of wire is usually the most desirable. This is called the wiring problem. The concept of MST (minimum spanning tree) evolves from modelling and solving this problem.

§9.2 Problem

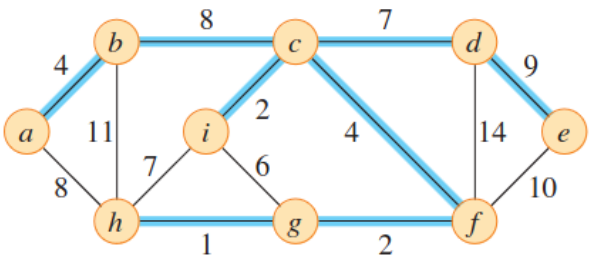
6 (Minimum Spanning Tree Problem)

To model the wiring problem, use a connected, undirected graph $G = (V, E)$, where V is the set of pins, E is the set of possible interconnections between pairs of pins, and for each edge $(u, v) \in E$, a weight $w(u, v)$ specifies the cost (amount of wire needed) to connect u and v. The goal is to find an acyclic subset $T \subset E$ that connects all of the vertices and whose total weight

$$w(T) = \sum_{(u,v)\in T} w(u, v)$$

is minimized. Since T is acyclic and connects all of the vertices, it must form a tree, which we call a spanning tree since it "spans" the graph G. We call the problem of determining the tree T the minimum-spanning-tree problem.

Example : The below figure represents a minimum spanning tree for a connected graph. The weights on edges are shown, and the blue edges form a minimum spanning tree. The total weight of the tree shown is 37. This minimum spanning tree is not unique: removing the edge (b, c) and replacing it with the edge (a, h) yields another spanning tree with weight 37.



§9.3 Generic Solution

The input to the MST is a connected, undirected graph $G = (V, E)$ with a weight function $w : E \to \mathbb{R}$. GENERIC-MST algorithm adopts a greedy strategy wherein it grows the MST one edge at a time. A set A of edges is managed with the following loop invariant: ***" Prior to each iteration, A is a subset of some minimum spanning tree "***

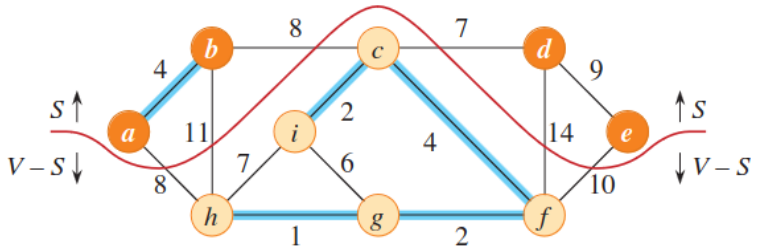
Algorithm GENERIC-MST(G, w)

```
1: A ← φ
2: while A does not form a spanning tree
3:         find an edge (u, v) that is safe for A
4:                 A = A ∪ (u, v)
5: return A
```

Terminology

- A ***cut*** $(S, V - S)$ of an undirected graph $G = (V, E)$ is a partition of V
- An edge $(u, v) \in E$ ***crosses*** the cut $(S, V - S)$ if one of its endpoints belongs to S and the other belongs to $V - S$
- A cut ***respects*** a set A of edges if no edge in A crosses the cut
- An edge is a ***light edge*** crossing a cut if its weight is the minimum of any edge crossing the cut. There can be more than one light edge crossing a cut in the case of ties. More generally, we say that an edge is a light edge satisfying a given property if its weight is the minimum of any edge satisfying the property
- Each step determines an edge (u, v) that the procedure can add to A without violating this invariant, in the sense that $A \cup \{(u, v)\}$ is also a subset of a minimum spanning tree. We call such an edge a ***safe edge*** for A, since it can be added safely to A while maintaining the invariant

Example : A cut $(S, V - S)$ of the graph in the below figure. Orange vertices belong to the set S, and tan vertices belong to $V - S$. The edges crossing the cut are those connecting tan vertices with orange vertices. The edge (d, c) is the unique light edge crossing the cut. Blue edges form a subset A of the edges. The cut $(S, V - S)$ respects S, since no edge of A crosses the cut.



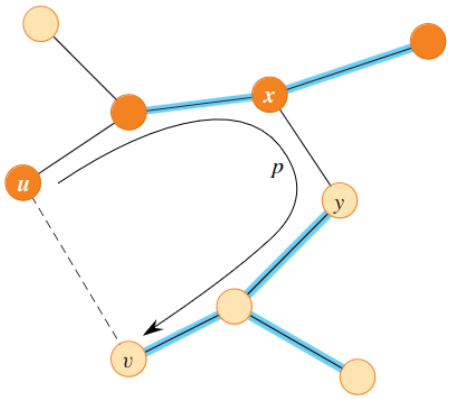
This generic algorithm uses the loop invariant as follows:

- **Initialisation :** After line 1, the set A trivially satisfies the loop invariant
- **Maintenance :** The loop in lines 2-4 maintains the invariant by adding only safe edges
- **Termination :** All edges added to A belong to a MST, and the loop must terminate by the time it has considered all edges. Therefore, the set A returned in line 5 must be a MST

The tricky part is, of course, finding a safe edge in line 3. One must exist, since when line 3 is executed, the invariant dictates that there is a spanning tree T such that $A \subseteq T$. Within the while loop body, A must be a proper subset of T , and therefore there must be an edge $(u, v) \in T$ such that $(u, v) \notin A$ and (u, v) is safe for A.

Theorem : *Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some MST for G, let $(S, V - S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing $(S, V-S)$. Then, edge (u, v) is safe for A.*

Proof : Let T be a MST that includes A, and assume that T does not contain the light edge (u, v), since if it does, we are done. We'll construct another MST T' that includes $A \cup \{(u, v)\}$ by using a cut-and-paste technique, thereby showing that (u, v) is a safe edge for A. The edge (u, v) forms a cycle with the edges on the simple path p from u to v in T , as the below figure illustrates. Since u and v are on opposite sides of the cut $(S, V - S)0$, at least one edge in T lies on the simple path p and also crosses the cut. Let (x, y) be any such edge. The edge (x, y) is not in A, because the cut respects A. Since (x, y) is on the unique simple path from u to v in T , removing (x, y) breaks T into two components. Adding (u, v) reconnects them to form a new spanning tree $T' = (T - \{(x, y)\}) \cup \{(u, v)\}$. We next show that T' is a MST. Since (u, v) is a light edge crossing $(S, V - S)$ and (x, y) also crosses this cut, $w(u, v) \leq w(x, y)$. Therefore, $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$. But T is a MST, so that $w(T) \leq w(T')$, and thus, T' must be a MST as well. It remains to show that (u, v) is actually a safe edge for A. We have $A \subseteq T'$, since $A \subseteq T$ and $(x,) \notin A$, and thus, $A \cup \{(u, v)\} \subseteq T'$. Consequently, since T' is a MST, (u, v) is safe for A. ■



As the method proceeds, the set A is always acyclic, since it is a subset of a MST and a tree may not contain a cycle. At any point in the execution, the graph $G_A = (V, A)$ is a forest, and each of

the connected components of G_A is a tree. (Some of the trees may contain just one vertex, as is the case, for example, when the method begins: A is empty and the forest contains $|A|$ trees, one for each vertex.) Moreover, any safe edge (u, v) for A connects distinct components of G_A, since $A \cup \{(u, v)\}$ must be acyclic. The while loop in lines 2-4 of GENERIC-MST executes $|V| - 1$ times because it finds one of the $|V| - 1$ edges of a MST in each iteration. Initially, when $A = \phi$;, there are $|V|$ trees in G_A , and each iteration reduces that number by 1. When the forest contains only a single tree, the method terminates.

Corollary : *Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some MST for G, and let $C = (V_C, E_C)$ be a connected component (tree) in the forest $G_A = (V, A)$. If (u, v) is a light edge connecting C to some other component in G_A , then (u, v) is safe for A.*

Proof : The cut $(C, V - C)$ respects A, and (u, v) is a light edge for this cut. Therefore, (u, v) is safe for A. ■

§9.4 Kruskal's algorithm

We have a $G = (V, E)$ be an undirected connected graph. Let L be the list of edges sorted in the increasing order of weight represented in the form of endpoints as (u, v) and V is the set of vertices. FIRST(L) denotes the the first element of L which is the edge with least weight. NEXT(e, L) implies the next edge in the list L. Analogously, the maximum spanning tree can be constructed by considering the edges sorted in the decreasing order of weight. Moreover, the collected set of edges forms a forest at any stage because we are ignoring those edges whichever form a cycle. The next-page includes the implementation-free generic pseudo-code for the algorithm as well as an example graph on which it is worked out step-by-step.

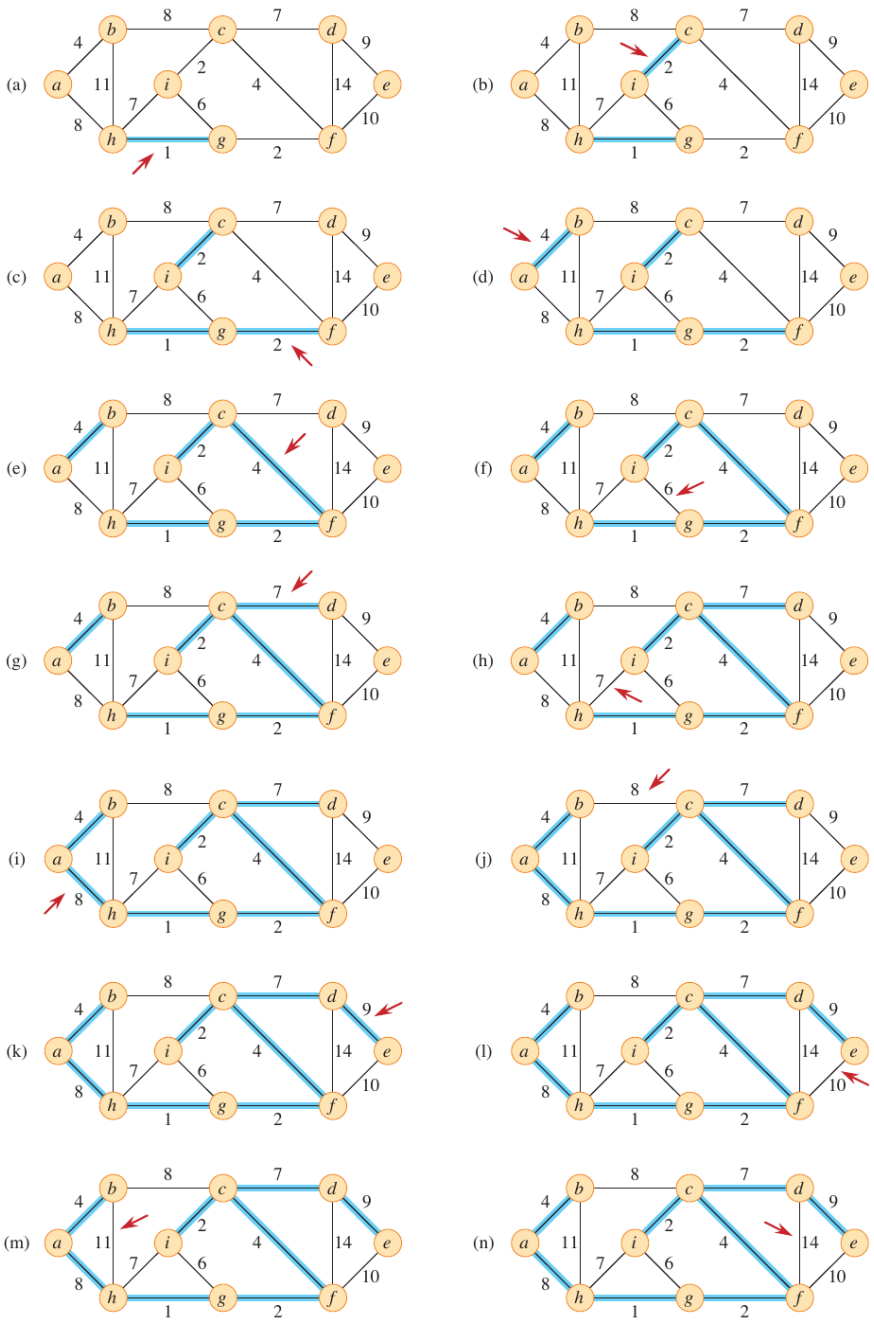
By first glance, it is not clear so as to why T is even a spanning tree, let alone the question of T being a MST. Hence, we give a formal proof of correctness now.

Proof of Correctness

Step 1 : T is a spanning tree due the following three stated reason.

Algorithm Kruskal's Algorithm - Generic(L)

```
1: T ← φ, e ← FIRST(L)
2: while (not end of L)
3:         if (e does not form a cycle with edges in T)
4:                 T = T ∪ {e}
5:         e = NEXT(e, L)
6: return T
```



- The first is that we clearly do not create any cycles, as we don't add any edges that would create any cycles.
- The second is that every vertex in G is in T. To prove this, let us assume the contrary: let v be the vertex that is not in T but is in G. Since G is connected, there must be some edges that are connecting to v in G. Note that we could have added an edge from the rest of T to v using any one of those edges without a cycle, so this is impossible.

- The third is that T must be connected. Proving this is the same as the proof of the last one, as if T is not connected, but G is connected, then we could have used an edge in G and added it to T without causing a cycle, giving us a contradiction.

Step 2 : Now that we proved that T is a spanning tree, we will now prove that it is a MST. We will prove this by induction. Let's say T isn't the minimum spanning tree, but T^* is. Then there is an edge $e \in T^*$, $e \notin T$. Consider $t \cup \{e\}$. Then there is a cycle C in T. In this cycle, with the exception of e itself, every edge will have a smaller weight than e (as otherwise, we would have chosen e to be in T instead). Since T^* doesn't have the cycle C (it's a tree), then there is an edge f that is in T but not in T^*. Thus, consider the tree $T^{'}$ by eliminating f from T and adding e. Now, note that $w(T^{'}) \geq w(T)$, and $T^{'}$ has more edges in common with T^* than T. If we keep doing this (exchanging edges) until we have T^*, we have that $w(T) \leq w(T^{'}) \leq w(T^{''}) \leq \cdots \leq w(T^*)$. But since $w(T^*)$ is the minimum weight, these inequalities must be equalities, and so T must be the same weight as t^*, and so it is also a MST.

Implementation

There are several data structures which can be used to implement. One such example using name array implementation has been discussed in 5.4.i. Here, we give an implementation using the abstract disjoint set data structure Union-Find. As a recap, in the pseudo-code, T contains the set of edges added so far. If G denotes the undirected connected graph under consideration then, $G.V$ denotes the set of vertices in G. We assume that edges are represented in the format $e = \{u, v\}$. L is list of edges in G sorted in increasing order of weight. The operations MAKE-SET, FIND-SET and UNION are as defined in chapter 5. Here, we assume that

- MAKE-SET(v) takes $O(t_m)$ time
- FIND-SET(v) takes $O(t_f)$ time
- UNION(x, y) takes $O(t_u)$ time.

Algorithm Kruskal's Algorithm - UNION FIND Implementation (V, L)

```
1: T ← φ
2: for (v ∈ G.V )
3:         MAKE-SET(v)
4: for (e ∈ L)
5:         if (FIND-SET(u) ≠ FIND-SET(v))
6:                 UNION(u, v)
7:                 T = T ∪ {e}
8: return T
```

Time Complexity

With the same notation as used in implementation above, we have

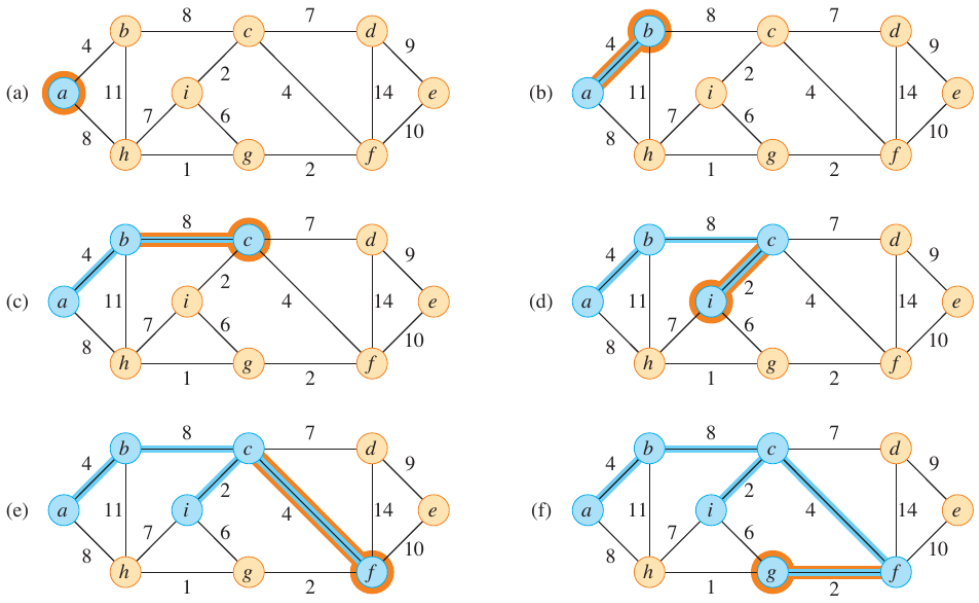
- Making the $|V|$ MST's takes $O(|V|.t_m)$ time

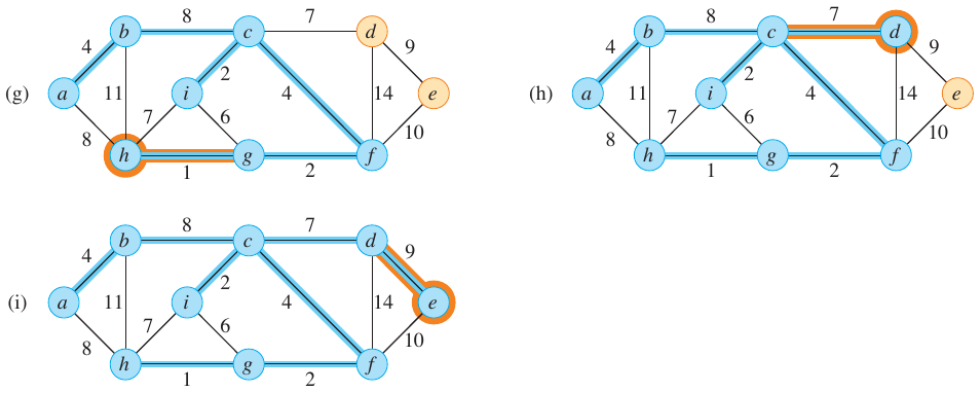
- Sorting the edges takes $O(|E|.\log(|E|))$ time
- The final loop takes $O(|E|.t_f + |V|.t_u)$ time

Hence, the algorithm as a whole takes $O(|V|.t_m + |E|.\log(|E|) + |E|.t_f + |V|.t_u)$. We shall measure these times with respect to the Link-by-Rank with Path compression technique described in 5.4.ii.5 since it is the asymptotically fastest implementation known. As per this, $O(t_m)$ is $O(1)$, $O(t_f)$ and $O(t_u)$ are both $O(\log(|V|))$. Now, since $|E| < |V|^2$, we have $O(\log(|E|)) = O(\log(|V|))$. Hence, we now have $O((|E| + |V|).\log(|V|))$ since $|V|$ is dominated by the $|V|.\log(|V|)$ term. Also because G is connected, we have $O(|V| + |E|) \sim O(|E|)$. Using all this, we can formally restate the running time of Kruskal's algorithm finally as $O(|E|.\log|V|)$. This shows that Kruskal's algorithm is better on sparse graphs because we don't have many edges. We can also that the running time is dominated actually the time required to sort the edges which is $O(|E|.\log(|E|)) \sim O(|E|.\log(|V|))$ because the core loop due to the heuristics actually takes only $O(|E|.\alpha(|E|, |V|))$ time where α is the inverse Ackermann function.

§9.5 Prim's Algorithm

Like Kruskal's algorithm, Prim's algorithm is a special case of the generic minimum-spanning-tree method. Prim's algorithm operates much like Dijkstra's algorithm for finding shortest paths in a graph, which we'll see later. Prim's algorithm has the property that the edges in the set A always form a single tree. The tree starts from an arbitrary root vertex r and grows until it spans all the vertices in V. Each step adds to the tree A a light edge that connects A to an isolated vertex - one on which no edge of A is incident. By the corollary proved earlier, this rule adds only edges that are safe for A. Therefore, when the algorithm terminates, the edges in A form a minimum spanning tree. This strategy qualifies as greedy since at each step it adds to the tree an edge that contributes the minimum amount possible to the tree's weight.





The execution of Prim's algorithm on an example graph is shown above. The root vertex is a. Blue vertices and edges belong to the tree being grown, and tan vertices have yet to be added to the tree. At each step of the algorithm, the vertices in the tree determine a cut of the graph, and a light edge crossing the cut is added to the tree. The edge and vertex added to the tree are highlighted in orange. In the second step (part (c)), for example, the algorithm has a choice of adding either edge (b, c) or edge (a, h) to the tree since both are light edges crossing the cut.

Algorithm MST_PRIM(G, w, r)

```
1:  for (each vertex u ∈ G.V)
2:          u.key = ∞
3:          u.π = NIL
4:  r.key = 0
5:  Q = φ
6:  for (each vertex u ∈ G.V)
7:          INSERT(Q, u)
8:  while (Q ≠ φ)
9:          u = EXTRACT_MIN(Q)
10:         for (each vertex v in G.Adj[u])
11:                 if ((v ∈ Q) and (w(u, v) < v.key))
12:                         v.π = u
13:                         v.key = w(u, v)
14:                         DECREASE_KEY(Q, v, w(u, v))
```

In the procedure MST_PRIM above, the connected graph G and the root r of the minimum spanning tree to be grown are inputs to the algorithm. In order to efficiently select a new edge to add into tree A, the algorithm maintains a min-priority queue Q of all vertices that are *not* in the tree, based on a *key* attribute. For each vertex v, the attribute $v.key$ is the minimum weight of any edge connecting v to a vertex in the tree, where by convention, $v.key = \infty$ if there is no such edge. The attribute $v.\pi$ names the parent of v in the tree. The algorithm implicitly maintains the set A from GENERIC_MST as $A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$ where we interpret the vertices in Q as forming a set. When the algorithm terminates, the min-priority queue Q is empty, and thus the minimum spanning tree A for G is $A = \{(v, v.\pi) : v \in V - \{r\}\}$.

Lines 1-7 set the key of each vertex to 1 (except for the root r, whose key is set to 0 to make it the first vertex processed), set the parent of each vertex to NIL , and insert each vertex into the min-priority queue Q. The algorithm maintains the following three-part loop invariant : Prior to each iteration of the ***while*** loop of lines 8-14

1. $A = \{(v, v.\pi) : v \in V - \{r\} - Q\}$
2. The vertices already placed into the minimum spanning tree are those in $V - Q$
3. For all vertices $v \in Q$, if $v.\pi \neq$ NIL, then $v.key < \infty$ and $v.key$ is the weight of a light edge $(v, v.\pi)$ connecting v to some vertex already placed into the minimum spanning tree.

Line 9 identifies a vertex $u \in Q$ incident on a light edge that crosses the cut $(V - Q, Q)$ (with the exception of the first iteration, in which $u = r$ due to lines 4-7). Removing u from the set Q adds it to the set $V - Q$ of vertices in the tree, thus adding the edge $(u, u.\pi)$ to A. The for loop of lines 10-14 updates the key and π attributes of every vertex v adjacent to u but not in the tree, thereby maintaining the third part of the loop invariant. Whenever line 13 updates $v.key$, line 14 calls DECREASE_KEY to inform the min-priority queue that v's key has changed.

The running time of Prim's algorithm depends on the specific implementation of the min-priority queue Q. You can implement Q with a binary min-heap, including a way to map between vertices and their corresponding heap elements. The BUILD_MIN_HEAP procedure can perform lines 5-7 in $O(V)$ time. In fact, there is no need to call BUILD_MIN_HEAP . You can just put the key of r at the root of the min-heap, and because all other keys are ∞, they can go anywhere else in the min-heap. The body of the ***while*** loop executes $|V|$ times, and since each EXTRACT_MIN operation takes $O(\lg V)$ time, the total time for all calls to EXTRACT_MIN is $O(V \lg V)$. The ***for*** loop in lines 10-14 executes $O(E)$ times altogether, since the sum of the lengths of all adjacency lists is $2|E|$. Within the ***for*** loop, the test for membership in Q in line 11 can take constant time if you keep a bit for each vertex that indicates whether it belongs to Q and update the bit when the vertex is removed from Q. Each call to DECREASE_KEY in line 14 takes $O(\lg V)$ time. Thus, the total time for Prim's algorithm is $O(V \lg V + E \lg V) = O(E \lg V)$, which is asymptotically the same as for our implementation of Kruskal's algorithm.

We can further improve the asymptotic running time of Prim's algorithm by implementing the min-priority queue with a Fibonacci heap. If a Fibonacci heap holds $|V|$ elements, an EXTRACT_MIN operation takes $O(\lg V)$ amortized time and each INSERT and DECREASE_KEY operation takes only $O(1)$ amortized time. Therefore, by using a Fibonacci heap to implement the min-priority queue Q, the running time of Prim's algorithm improves to $O(E + V \lg V)$.

10 Amortized Analysis

§10.1 Introduction

In an amortized analysis, the time required to perform a sequence of data-structure operations is averaged over all the operations performed. Amortized analysis can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a single operation within the sequence might be expensive. Amortized analysis differs from average-case analysis in that probability is not involved; an amortized analysis guarantees the average performance of each operation in the worst case. We will now give an example of accounting method in the next section.

§10.2 Accounting method

In the accounting method of amortized analysis, we assign differing charges to different operations, with some operations charged more or less than they actually cost. The amount we charge an operation is called its amortized cost. When an operation's amortized cost exceeds its actual cost, the difference is assigned to specific objects in the data structure as credit. Credit can be used later on to help pay for operations whose amortized cost is less than their actual cost. Thus, one can view the amortized cost of an operation as being split between its actual cost and credit that is either deposited or used up.

One must choose the amortized costs of operations carefully. If we want analysis with amortized costs to show that in the worst case the average cost per operation is small, the total amortized cost of a sequence of operations must be an upper bound on the total actual cost of the sequence. This relationship must hold for all sequences of operations. If we denote the actual cost of the i^{th} operation by c_i and the amortized cost of the i^{th} operation by $\tilde{c}_i$, we require

$$\sum_{i=1}^{n} \tilde{c}_i \geq \sum_{i=1}^{n} c_i$$

for all sequences of n operations. The total credit stored in the data structure in is the difference between the total amortized cost and the total actual cost, or

$$\sum_{i=1}^{n} \tilde{c}_i - \sum_{i=1}^{n} c_i$$

. By the above inequality, the total credit associated with the data structure must be non-negative at all times. If the total credit were ever allowed to become negative (the result of undercharging early operations with the promise of repaying the account later on), then the total amortized costs incurred at that time would be below the total actual costs incurred; for the sequence of

operations up to that time, the total amortized cost would not be an upper bound on the total actual cost. Thus, we must take care that the total credit in the data structure never becomes negative.

As an example, we analyze stacks that have been augmented with a new operation. Thus, we have the following operations:

- PUSH (S, x) pushes object x onto stack S
- POP(S) pops the top of stack S and returns the popped object
- MULTIPOP (S, k), which removes the k top objects of stack S, or pops the entire stack if it contains fewer than k objects

Let us analyze a sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack. The worst-case cost of a MULTIPOP operation in the sequence is $O(n)$, since the stack size is at most n. The worst-case time of any stack operation is therefore $O(n)$, and hence a sequence of n operations costs $O(n^2)$, since we may have $O(n)$ MULTIPOP operations costing $O(n)$ each. Although this analysis is correct, the $O(n^2)$ result, obtained by considering the worst-case cost of each operation individually, is not tight.

If we let s to be the number of stack objects at any point, then we use the following cost table for analysis :

Operation	Actual Cost	Amortized Cost
PUSH	\$1	\$2
POP	\$1	0
MULTIPOP	\$min(k, s)	0

We shall now show that we can pay for any sequence of stack operations by charging the amortized costs. Lets consider the elements as plates in a cafeteria. When we push a plate on the stack, we use \$1 to pay the actual cost of the push and are left with a credit of \$1 (out of the \$2 charged), which we put on top of the plate. At any point in time, every plate on the stack has a dollar of credit on it.

The dollar stored on the plate is prepayment for the cost of popping it from the stack. When we execute a POP operation, we charge the operation nothing and pay its actual cost using the credit stored in the stack. To pop a plate, we take the dollar of credit off the plate and use it to pay the actual cost of the operation. Thus, by charging the PUSH operation a little bit more, we needn't charge the POP operation anything.

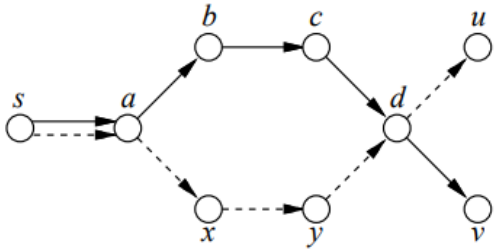
Moreover, we needn't charge MULTIPOP operations anything either. To pop the first plate, we take the dollar of credit off the plate and use it to pay the actual cost of a POP operation. To pop a second plate, we again have a dollar of credit on the plate to pay for the POP operation, and so on. Thus, we have always charged enough up front to pay for MULTIPOP operations. In other words, since each plate on the stack has 1 dollar of credit on it, and the stack always has a nonnegative number of plates, we have ensured that the amount of credit is always nonnegative. Thus, for any sequence of n PUSH, POP , and MULTIPOP operations, the total amortized cost

is an upper bound on the total actual cost. Since the total amortized cost is $O(n)$, so is the total actual cost.

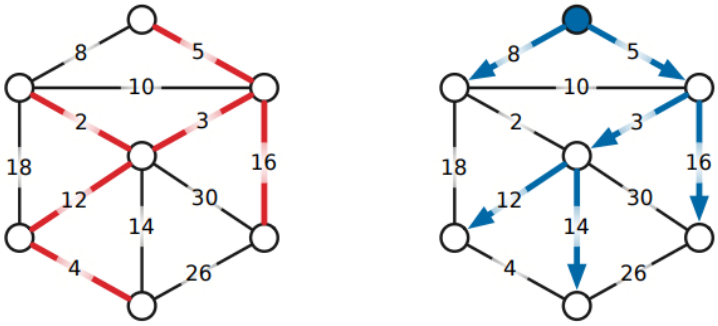
11 Shortest Path Problem

§11.1 Introduction

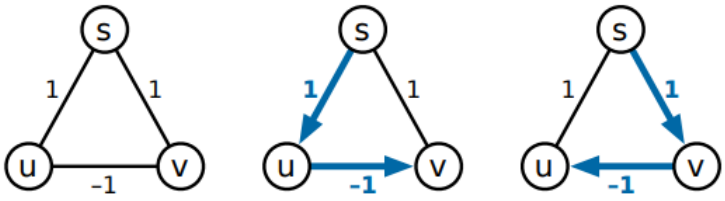
We have seen how to solve minimum spanning tree problem. We shall now look at another type of problem - the shortest path problem. As the name indicates, the most basic version involves finding the shortest distance between two given vertices, say a source vertex s and a target vertex t. Almost every algorithm known for solving this problem actually solves (large portions of) the following more general single source shortest path or SSSP problem : Find the shortest path from the source vertex s to every other vertex in the graph. This problem is usually solved by finding a shortest path tree rooted at s that contains all the desired shortest paths. It's not hard to see that if shortest paths are unique, then they form a tree, because any subpath of a shortest path is itself a shortest path. If there are multiple shortest paths to some vertices, we can always choose one shortest path to each vertex so that the union of the paths is a tree. If there are shortest paths to two vertices u and v that diverge, then meet, then diverge again, we can modify one of the paths without changing its length so that the two paths only diverge once.



Here, if $s \to a \to b \to c \to d \to v$ and $s \to a \to x \to y \to d \to u$ are shortest paths, then $s \to a \to b \to c \to d \to u$ is also a shortest path. In short, although they are both optimal spanning trees, shortest-path trees and minimum spanning trees are very different creatures. Shortest-path trees are rooted and directed; minimum spanning trees are unrooted and undirected. Shortest-path trees are most naturally defined for directed graphs; only undirected graphs have minimum spanning trees. If edge weights are distinct, there is only one minimum spanning tree, but every source vertex induces a different shortest-path tree; moreover, it is possible for every shortest path tree to use a different set of edges from the minimum spanning tree. The below figure shows a minimum spanning tree and a shortest path tree (rooted at the top vertex) of the same undirected graph.



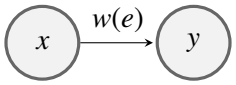
Remark 3 — Throughout this chapter, we will explicitly consider only directed graphs. All of the algorithms described in this chapter also work for undirected graphs with some minor modifications, but only if negative edges are prohibited. Dealing with negative edges in undirected graphs is considerably more subtle. We cannot simply replace every undirected edge with a pair of directed edges, because this would transform any negative edge into a short negative cycle. Subpaths of an undirected shortest path that contains a negative edge are not necessarily shortest paths; consequently, the set of all undirected shortest paths from a single source vertex may not define a tree, even if shortest paths are unique.



We also have another version called all-pairs shortest path problem (APSP), wherein given n vertices, we find the shortest paths between all $\binom{n}{2}$ pairs. Firstly, we will consider how to tackle the SSSP type.

§11.2 Statement

Consider a directed graph $G = (V, E)$ with an associated weight function w. We assume that an edge $e \in E$ is represented in terms of its endpoints, say x and y. Since, it is a directed graph, we use the notation $e = (x, y)$ to indicate that the edge e comes out from the vertex x and goes into vertex y with a weight $w(e)$.

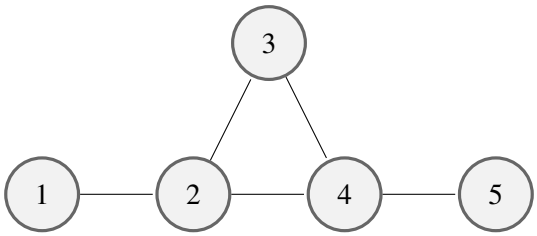


If we don't have any weights associated, then we take the weights of all edges as 1 in which case the length of path is the number of edges itself constituting the path where length is defined

as follows. If $p(u, v)$ is a directed path from $u = v_0 \to v_1 \to \cdots \to v_{k-1} \to v_k = v$, then the $v_1, \ldots, v_{k-1}$ are considered as the intermediate vertices for a path and its length l is defined as $l(p(u, v)) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$. If p^* is physically the shortest path, then we denote its length by $\delta(u, v)$ i.e., $\delta(u, v) = l(p^*(u, v))$.

We shall now digress and look into detail as to what kind of values can be taken up by weights. We shall only consider those weights which are either positive or negative integers including zero. For rational numbers, we either scale it down to the nearest integer or bound it by two integers. In case of real numbers, since computers can only handle numbers up to a finite precision, the case can be handled similar to the rational numbers case. As a matter of fact, dealing with reals is tricky for the arithmetic involved is quite different for example, well-ordering principle is present only in the case of integers but not for reals. Hence, we make the switch to rationals or integers.

We shall make another digression and have a look at usage of the word ***path*** which was made a complete mess by computer scientists. Mathematicians have distinctly defined three terms - namely walk , path and trail - for graphs. In the following figure, if we traverse vertices in the order $1 \to 2 \to 3 \to 4 \to 5$, then it is called a path. However, if we traverse the vertices in the order $1 \to 2 \to 3 \to 4 \to 2 \to 3 \to 4 \to 5$, then it is called a walk but not a path. Hence, in a path, the number of edges or vertices can be repeated any number of times, while not in a path. It is not difficult to see that each walk has a path as a subsequence, which can be obtained by short circuiting the last occurrence of a repeated vertex. For example, in the walk, we see that vertices 2,3 and 4 have repeated occurrences however the vertex occurring last is 4. Hence, we short circuit the red part from the walk - $1 \to 2 \to 3 \to 4 \to 2 \to 3 \to 4 \to 5$. In computer science, the words path and walk are used interchangeably, hence it is our responsibility to take the correct interpretation at each such occurrence.



To tackle the problem, given this setting, until now no polynomial time solution has yet been discovered. However, for most practical instances, such as considering the case of only positive edge weights, polynomial time solutions do exist. However, when considering negative weights, the problem crucially arises due to the presence of negative cycles where a cycle is defined as any sequence of vertices with the same start and end vertex. Hence any cycle is of the form $v_0 \to v_1 \to \cdots \to v_k \to v_1$ and if $l(c)$ is greater than zero, then it is a positive cycle ; if $l(c)$ is zero, it is a zero cycle and if $l(c)$ is less than zero, it is a negative cycle. We can now see that if a graph has a negative cycle, then the length of a shortest walk between two vertices is ill-defined because it can be made $-\infty$, by going around the negative cycle. However, a shortest path is still well defined. We can also see that the number of walks is ∞ while the the number of paths is

finite, in fact it can be at most $(n-2)!$ given n vertices.

Historically, when the assumption that the graph has no negative or zero cycles was made in the initial stages, researchers were puzzled so as to was the problems inherently difficult or it was just that they weren't able to solve it, until the early seventies, when Hooke's theorem and other parallel results led to the development of notion of an inherent difficulty and NP-incompleteness. In SSSP, we assume that a path exists between two vertices, else we define its length as ∞ and subsequently, keep updating the lengths. The technique involved is the method of iterated improvement which is different from incremental design based on the principle of building a partial solution for a sub instance of the problem and developing it later to a bigger instance.

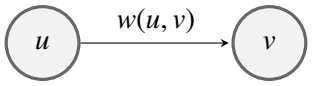
For any vertex v, we start off with an initial assumption, say $d(v)$ which is the length of some path from s to v. Moreover, due to assumption that the graph has no negative or zero cycles, the length of shortest path from s to s itself is zero. Thus initially we have $d(s) = 0$ and $d(v) = \infty$ for $v \neq s$. Here, ∞ is chosen because it acts as an identity for minimum operation, since if any other length is discovered, then being lesser than it, would be taken by the algorithm. It keeps constantly updating this, so called distance vector, until we reach the stage $d(v) = \delta(v)$ and $d(s) = \delta(s)$. If there is no path, then we wont touch ∞ or in other words, it wont be updated.

§11.3 SSSP

In SSSP, as discussed in the last section, we are given a directed graph $G = (V, E)$ where each edge $e \in E$ has an associated weight, namely $w(e)$. We then consider a source vertex $s \in V$. Our goal is to output $n-1$ shortest paths from s to v, $\forall\ v \in V - \{s\}$, where n is the number of vertices. We note that from now on, δ_v is the length of the shortest path from s to v and $\delta_s = 0$ by assuming that G has no negative or zero cycles. We will now obtain a set of relations, better called functional equations which shall be solved analytically or theoretically. Note that an analytical approach is always more elegant but may be mathematically difficult to solve. We shall now state two properties with proofs.

PROPERTY I : *If* $(u, v) \in E$, *then* $\delta_u + w(u, v) \geq \delta_v$

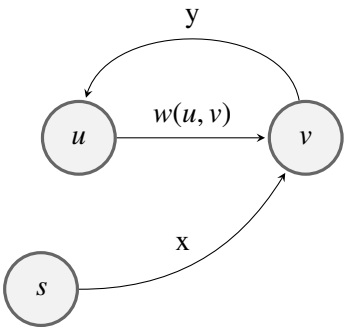
One may be tempted to say that the LHS represents the length of some path from s to v which is obviously greater than or equal to the RHS because δ_v represents the shortest path from s to v. However, this is not trivially true as it seems to be.



In fact we made no use of the assumption that there are no negative or zero cycles. So, now we present a formal proof which consists of two cases, namely

- **Case I :** *If the shortest path to u passes through v*

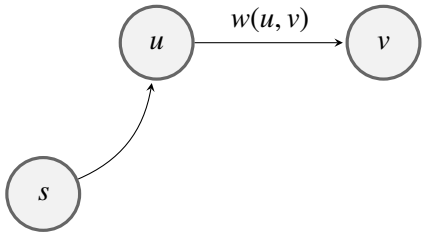
In this case, the scenario looks as follows, where the curved paths could be either simply one single directed edge or a combination of several such directed edges between the intermediate vertices.



We see that in this case adding the edge (u, v) results in a walk but not a path. Moreover, the length of sub-path from s to v is taken as x i.e., $l(p(s, v)) = x$ and $l(p(v, w)) = y \implies x + y = \delta_u$. We also have that $x \geq \delta_v$. For proving, we assume for the sake of contradiction that $\delta_u + w(u, v) < \delta_v \implies x + y + w(u, v) < \delta_v \leq x$. Cancelling x on both sides, yields $y + w(u, v) \leq 0$. If we denote the cycle formed by the path $p(v, u)$ and the edge (u, v) by c and its length as $l(c)$, then we have that $l(c) = w(u, v) + y$ which is less than or equal to zero contradicting the assumption that there are no negative or zero cycles.

- **Case II :** *If the shortest path to u doesn't pass through v*

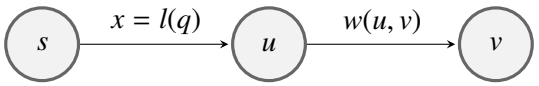
  In this case, the scenario looks as follows. Here, we can clearly see that by adding the edge (u, v), we obtain a path from s to v. Its length then obviously has to be greater than or equal to δ_v because it is length of the shortest path from s to v.



Hence, we have shown that $\forall\ (u, v) \in E,\ \delta_u + w(u, v) \geq \delta_v$. Note that this is not just one inequality but a collection because the number of inequalities, we have proven is equal to the in-degree of G and not just one. The next property tries to establish equality in atleast one case.

PROPERTY II : There exists $(u, v) \in E$, such that $\delta_u + w(u, v) = \delta_v$

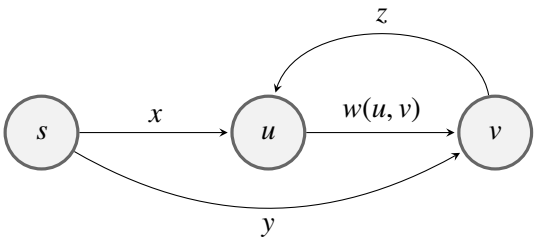
Let us now consider $p^*(s, v)$ i.e., the shortest path from s to v with length equal to δ_v i.e., $l(p^*(s, v)) = \delta_v$. Say that the last intermediate vertex of p^* is u. Then we shall prove that the subpath of $p^*(s, v)$ from s till u, say q is itself $p^*(s, u)$ i.e., $l(q) = l(p^*(s, u)) = \delta_u$.



Hence, we have two situations depending on whether q is $p^*(s,u)$ or not. If q is $p^*(s,u)$, then we are done. Else, we again have two cases, namely

- **Case I :** *If the shortest path to u passes through v*

  In this case, the scenario looks as follows where $l(p^*(s,u)) = y + z$. We first of all observe that $x > y + z$. Now, we see that y is the length of a random path from s to v. Hence, we also have that $y \geq \delta_v = x + w(u,v)$. Adding the two inequalities, followed by cancellation of x and y on both sides, we obtain that $0 > z + w(u,v)$. However, note that $z + w(u,v)$ is the length of the cycle formed by the edge (u,v) and the path p from v to u whose length is z. This contradicts the assumption that G has no negative cycles. The only assumption which could have gone wrong was $q \neq p^*(s,u)$. We have thus shown in this case that $q = p^*(s,u)$.



- **Case II :** *If the shortest path to u doesn't pass through v*

  If $p^*(s,u)$ doesn't pass through v, then $p^*(s,u) + (u,v)$ is a path from s to v. Hence, $\delta_v = x + w(u,v) > \delta_u + w(u,v)$. This implies that I have now found a path $(p^*(u,v) + (u,v))$ whose length $(\delta_u + w(u,v))$ is lesser than δ_v contradicting the fact that δ_v is the length of the shortest path to v from s. The only assumption which could have gone wrong was $q \neq p^*(s,u)$. We have thus shown in this case as well that $q = p^*(s,u)$.

Note that this property only implies that there is no other shorter path than this but not that there are no other shortest paths possible from s to u. We have hence proved that $\exists\ (u,v) \in E$ such that $\delta_u + w(u,v) = \delta_v$. Combining these two inequalities, we can say

$$\delta_v = \min_{(u,v)\in E}\{\delta_u + w(u,v)\}$$

Note that only the first inequality alone would not have been sufficient to conclude this. We have thus obtained a functional equation. Now, what Bellman did, was that he defined a variable x_v for each vertex such that $x_s = 0$ and $x_v = \min_{(u,v)\in E}\{x_u + w(u,v)\}$. This looks similar to the iterative method of solving linear equations like Jacobi's method where the i-th equation is defined in terms of all the previous $(i-1)$th quantities. Moreover, if we set $w(u,v) = \infty$ if $(u,v) \notin E$, then the above inequality can be restated as follows,

$$\delta_v = \min_{u\neq v}\{\delta_u + w(u,v)\}$$

The advantage by representing it in this format is that now, for every vertex v the RHS represents the minimum of a collection of $(n-1)$ quantities, unlike the earlier version, wherein it represented the minimum of a collection of in-degree(v) number of quantities.

Aside 1 (JACOBI METHOD)

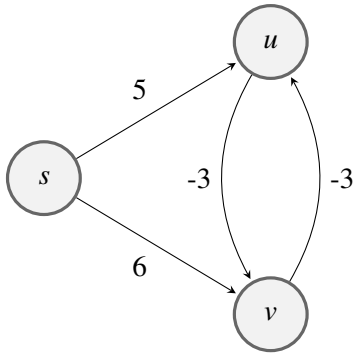
Let $A\vec{x} = \vec{b}$ be a system of n linear equations, where

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix}, \vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \vec{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

When A and $\vec{b}$ are known, but $\vec{x}$ is unknown, we can use the Jacobi method to approximate $\vec{x}$. The vector $\vec{x}^{(0)}$ denotes our initial guess for ($\vec{x}$). We denotes $\vec{x}^{(k)}$ as the k-th approximation or iteration of $\vec{x}$ and $\vec{x}^{(k+1)}$ is the next or $(k+1)$-th iteration of $\vec{x}$. The element-based formula for each row i is thus,

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left[b_i - \sum_{i \neq j} a_{ij} x_j^{(k)} \right], i = 1, 2, \dots, n$$

Note that even though Gaussian elimination method is exact but its complexity is $O(n^3)$ while in Jacobi method, the complexity of each iteration is only $O(n^2)$. Moreover, in Jacobi method we also have faster convergence as well as control over the error involved. As an aside, we see that the cut-and-paste argument used in proving property II would not be valid if we had a negative cycle as we shall see using the following simple example involving just three vertices. Here, the shortest path from s to u is $s \to v \to u$ and $\delta_u = 6 - 3 = 3$. However, $s \to v$ is not the shortest path from s to v instead it is $s \to u \to v$ and $\delta_v = 5 - 3 = 2$.

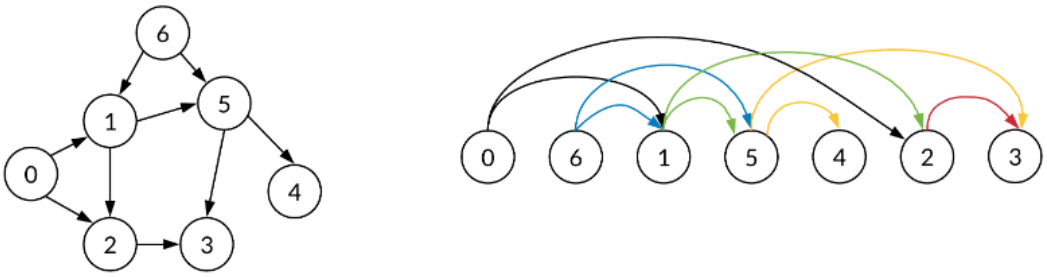


However, since we don't have negative cycles, we can safely claim that the optimal solution is made up optimal sub-solutions. This is in fact known as the principle of optimality better stated as *"An optimal policy has the property that whatever the initial state and initial decisions are, the remaining decisions must constitute an optimal policy with regard to the state resulting from*

the first decisions.". If this principle is satisfied, then it is guaranteed that the we can develop a recurrence relation which could be of various forms like for example tail recursion or in the case of divide and conquer, where we break the problems into two sub-problems which can be solved independently. In order to solve such recurrences, Bellman had developed ***dynamic programming*** which basically maintains a table containing solutions of all the problems. This was necessary because in certain instances going from a recursive formulation to a recursive program prove to be disastrous as it often yielded exponential solutions as in the current case. In fact, the method described by Bellman and Ford was iterative in nature as shown above.

There are certain physical instances, as we shall see, which can be solved directly with the iterative analogue for the current problem, namely

- *G is a DAG :* When G is a directed acyclic graph (DAG), then we have no cycles at all. However, we have a topological sorting which is nothing but a linear ordering of the vertices of a DAG such that for every directed edge (u, v), vertex u comes before vertex v in the ordering. In other words, it is a way to order the vertices of a DAG such that there are no directed cycles. As an example, consider the following figure, where on the left we have a DAG and on the right we have its corresponding topologically sorted version. With this ordering, the adjacency matrix of a DAG can be made upper triangular and by applying Gaussian elimination, we can easily solve the problem now.



- *G has only positive edge weights :* We shall see this case in particular when we will be studying about Dijkstra's algorithm in the subsequent sections.

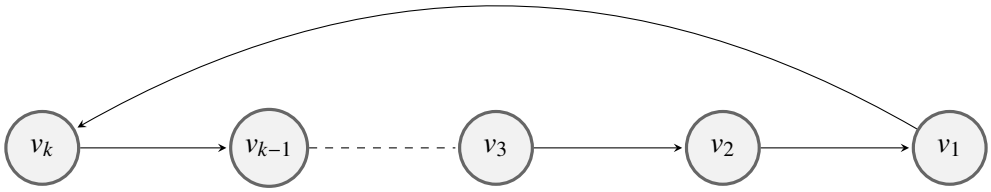
We can also see that there is a particular class of graphs which can be solved peacefully using the minimum formulation. If a graph doesn't belong to this class, then we go to the iterative method. Moreover, we perform the procedure until we reach the solution i.e., $x_s = 0$ and $x_v = \delta_v \; \forall \; v \in V - \{s\}$. However, how can we be sure that the procedure indeed converges, and if it does actually, then to the solution itself or not. For example, if while solving a quadratic equation we get two roots - one of which is positive and one is negative - then we can discard the negative root and say that the positive one is the answer. However, if the procedure gives for example, two positive solutions, then how do we decide which one is correct. As an answer to all such questions, Bellman gave an elegant proof that his equations always have a unique solution.

He did so by building what is known as a solution graph wherein for a vertex v, he considered the vertex u satisfying $x_v = x_u + w(u, v)$ or $\delta_v = \delta_u + w(u, v)$. If there are several such minimal

u's, then choose one u and fix it. In this way, we are always fixing only one incoming edge and the resulting graph has $(n-1)$ edges. In fact, the solution graph satisfies the following properties ,

- A solution graph has $(n-1)$ edges
- The source vertex s has no incoming edge
- Every vertex v has exactly one incoming edge
- A solution graph has no cycles

Assume that the solution graph has a cycle for the sake of obtaining a contradiction, then we can show that weight of the cycle is zero as follows. Lets use the following notation depicted in the diagram below to avoid confusion.



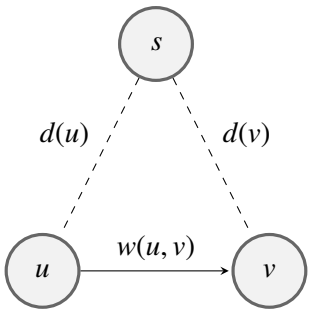
Then, based on our understanding, we write the following equations. Let us denote by Now, by summing up all these equations and cancelling terms, we end up on the RHS with $w(v_1, v_k) + w(v_2, v_1) + w(v_3, v_2) + \ldots w(v_k, v_{k-1}) = 0$. This is nothing but the weight of the cycle formed by the edges $(v_k, v_{k-1}), \ldots, (v_3, v_2), (v_2, v_1)$ and (v_1, v_k). Therefore, we have obtained a contradiction because G can't have zero-weight cycles.

$$
\begin{aligned}
\delta_{v_1} &= \delta_{v_2} + w(v_2, v_1) \\
\delta_{v_2} &= \delta_{v_3} + w(v_3, v_2) \\
&\vdots \\
\delta_{v_{k-1}} &= \delta_{v_k} + w(v_k, v_{k-1}) \\
\delta_{v_k} &= \delta_{v_1} + w(v_1, v_k)
\end{aligned}
$$

Due to this reason, this is also called a solution tree. Moreover, since all the edges are directed away from the root s to all other vertices, this is an out tree and a path between between any vertex v and s in the solution tree is in fact the shortest path. Since, there is no cycle we are guaranteed to hit the vertex s using at max $(n-1)$ edges. Moreover, we then have $\delta_{v_1} = w(v_k, v_{k-1}) + \cdots + w(v_2, v_1)$, if we consider v_k as the source s because then $\delta_{v_k} = 0$. However, note that this is giving the list

of vertices occurring in the shortest path from the source to a vertex in the reverse order. We now introduce a term ***predecessor***, written in short as pred meaning that if pred(v) = u, then the solution graph has the edge (u, v) with weight $w(u, v)$. In this notation, the predecessor of source is assumed to be null or undefined. Hence, we are obtaining the vertices in the order v, pred(v), pred(pred(v)). . . s. To obtain in the reverse order, we can develop a stack with v as the first element and then keep pushing pred(v) until we reach s, followed by popping the elements.

Thus, for every vertex we are maintaining only one piece of information which is the predecessor and the labour required is in proportion to the length of path. Hence, the output would involve the length of shortest path from s to v along with its predecessor whose value we save.

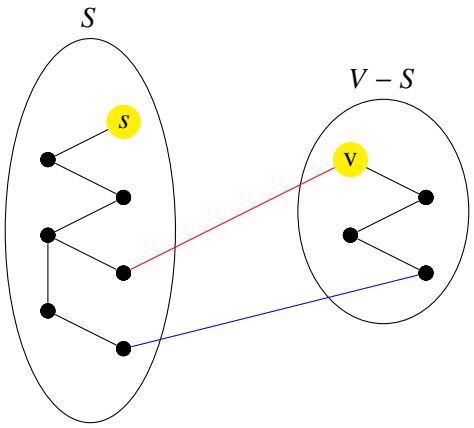


Thus, if we start off with estimates $d(u)$ and $d(v)$ which are lengths of some paths, we perform the following steps If after parsing through all vertices, no more edge is causing any improvement,

```
1: if d(u) + w(u, v) < d(v)
2:         d(v) = d(u) + w(u, v)
3:         pred(v) = u
4: else
5:         do nothing
```

then we can say that $d(v)$ is indeed δ_v. Thus, the convergence condition is when no more edge causes any further improvement and the number of parses to make the solution stable is $(n - 1)$.

Let us now revisit the case we promised earlier to deal with i.e., when G has only positive edges in other words $\forall\ (u, v) \in E$, we have $w(u, v) > 0$. As seen before, for any stage we see that the shortest distances of some vertices are known. Lets call this set S and the set of vertices whose shortest distances are unknown as $V - S$. We now consider a path from s to v as special if all its intermediate vertices / internal nodes are in S. Hence, with this notation, we see that $d(v)$ is the length of the shortest special path for each vertex $v \in V - S$. In the below figure, the red path is special but not the blue one.



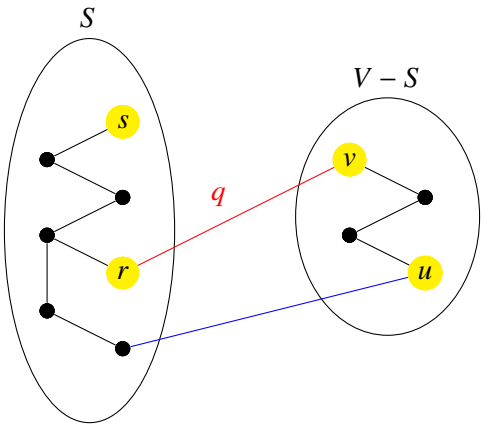
An alternate starting condition could as well be $s = \{\phi\},\ V - S = V$ along with $d(s) = 0,\ d(v) = \infty$ and pred(v) = NULL.

§11.3.i Dijkstra's Algorithm

Here, we initialize $S = \{s\}$, pred(v)= s if $w(s, v) < \infty$ and while $(V - S) \neq \phi$, we perform the following steps, namely

- Let u be a vector in $V - S$ such that $d(u) = \min_v\{d(v) : v \in V - S\} \implies d(u) = \delta_u$
- Move u to S
- Now we update lengths. For each edge $(u, v) \in E$, $v \in V - S$, if $d(u) + w(u, v) < d(v)$, then we set $d(v)$ to $d(u) + w(u, v)$ and pred$(v) = u$

What we need to show is that $d(u) = \delta_u$, so that we can safely move u to S. Lets prove this by contradiction. Consider the following figure as a reference.



Assuming $l(q) < d(u)$, then we have

$$\begin{aligned} l(q) &= l(q(s,r)) + w(r,v) + l(q(v,u)) \\ l(q) &\geq l(q(s,r)) + w(r,v) \\ &\geq \delta_r + w(r,v) \\ &= d(r) + w(r,v) \\ &\geq d(v) \rightarrow \text{length of } d(v) \text{ when } v \text{ added to } s \\ &\geq d(v) \text{ currently} \\ \therefore d(u) > l(q) \geq d(v) &\implies d(u) > d(v) \end{aligned}$$

However, this is a contradiction since $d(u)$ was chosen as the minimum of the set $d(v) : v \in V - S$. Hence, $d(u) = \delta_u$ and the algorithm produces the correct solution only for $w(u,v) > 0$. This fails for graphs with negative or zero cycles.

Implementation

Initially at the time of discovery (around the 1950's) of the algorithm, heaps weren't yet known. Thus, an array-based implementation was used which was good for dense graphs as we shall see. However, subsequently when heaps were discovered - they were used for implementation which was good only for sparse graphs. The current version is to use the Fibonacci heaps which are good for both - dense and sparse graphs.

Time Complexity

We have seen three methods of implementation. Lets see the time complexity for each method.

- *Array :* Using arrays, we see that the time complexity $O(n^2 + m)$ where m is the number of edges. For sparse graphs, the complexity becomes just $O(n^2)$. Even for dense graphs, when $m = n^2$, the total complexity is still $O(n^2)$.
- *Heap :* We split the analysis into two parts. First part involves finding the min and deleting it operation, which we perform in each operation. The cost of finding min and del*(min)* operation for heaps is $O(\log n)$, hence the total complexity of this step is $O(n \log n)$ because we have n iterations. However, at each stage we see that the value of $d(v)$ is being decreased and hence this calls for a restructuring of the heap. In the worst case - we may have to perform m del(*key*) operations each of complexity $O(\log n)$. Thus the overall complexity becomes $O((n + m)\log n)$. This is good for sparse graphs as the complexity is $O(n \log n)$, while for dense graphs when $m = n^2$, the complexity becomes $O(n^2 \log n)$.

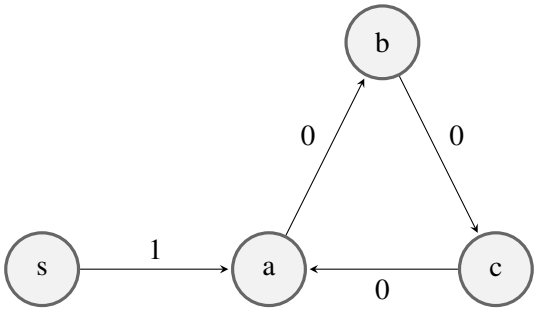
However when Fibonacci heaps were discovered in 1963, using amortized analysis we saw that the cost of m del(*key*) operations is $O(m)$. Hence, the total complexity in this case becomes $O(n \log n + m)$ which is good for both sparse graphs and dense graphs. However, Dijkstra's algorithm causes a problem if its predecessor graph contains a zero cycle in which case we can never reach the source vertex by predecessor chasing. Moreover, it can be shown with an easy counterexample that it fails for negative edges. In principle, its behaviour is unpredictable when there are negative edges i.e., there is no guarantee is wont work. We shall now see another algorithm developed by two giants or thought leaders namely Bellman and Ford, which deals withe case when negative edges may be present but not negative cycles.

§11.3.ii Bellman-Ford Algorithm

The formalisation is same as before, namely we are given a directed graph = (V, E) with an associated weight function $w(e)$ and a special vertex s. We are interested in finding δ_v i.e., the shortest path to s for all vertices $v \in V$. To solve this, Bellman adopted a functional equation approach and gave the following equations, which are satisfied by the lengths of the shortest paths for a graph.

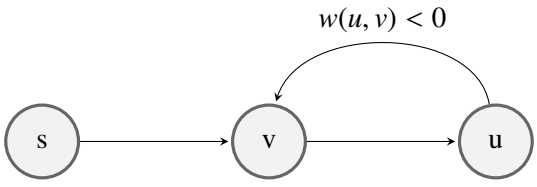
$$x_s = 0$$
$$x_v = \min_{u \neq v}\{x_u + w(u, v)\}$$

The set of x values satisfying these equations happen to be the lengths of the shortest paths itself when there are no negative or zero cycles. However, in their presence, the sets of x values obtained by solving the equations need not be necessarily be the lengths of shortest paths as we shall see with an example.



For this graph, our Bellman equations become $x_s = 0, x_a = \min\{1, x_c\}, x_b = x_a, x_c = x_b$ We can see that the set of lengths of shortest paths for the above graph is $\delta_a = \delta_b = \delta_c = 1$. This set of values satisfies the above equations indeed. However, we also see that any $x_c < 1$, also satisfies the above equations in which case we have an infinite set of values such that $x_a = x_b = x_c = k < 1$. What Bellman did was that he proved the uniqueness of the solution in the case when there are no negative or zero cycles. We now present the proof of uniqueness as presented in the paper published by Bellman himslef in 1958, which is skipped in most textbooks these days.

The $d(v)$ in Dijkstra's algorithm has an interpretation namely length of the shortest special path. However, when negative edges are included, $d(v)$ need not necessarily mean the same thing. In fact, when negative edges are there, we may get a walk as well i.e., it may happen that $w(u, v) < 0$ and $d(v) > d(u) + w(u, v)$ in which case setting $d(v) = d(u) + w(u, v)$ indeed sets a lower value to $d(v)$ but it need not represent the length of a path as indicated in the following figure,



Aside 2 (Proof of Uniqueness of Bellman's equations)

Assume that the sets $P = \{p_v\}$ and $Q = \{q_v\}$ are two solutions with $p_s = q_s = 0$ and let k be a vertex for which $p_k - q_k$ achieves its maximum value. Then

$$p_k = \min_{u \neq k}\{p_u + w(u,k)\}$$
$$q_k = \min_{u \neq k}\{q_u + w(u,k)\}$$

Let the minimum in the first equation be assume for $u = r$ and in the second equation for $u = s$. It is clear that since $w(u,k) > 0$ for all u, k that $r \neq k, s \neq k$. Then we have the equalities and in-equalities,

$$p_k = p_r + w(r,k) \leq p_s + w(s,k)$$
$$q_k = q_s + w(s,k) \leq q_r + w(r,k)$$

These lead to

$$p_r - q_r \leq p_k - q_k \leq p_s - q_s$$

Since k was a vertex for which $p_k - q_k$ achieved its maximum value, we must have $p_k - q_k = p_s - q_s$ which can only be true if we have $p_k = p_s + w(s,k)$. Now repeat this procedure for the pair $\{p_s, q_s\}$. It follows from the foregoing argument that there must be another pair $\{p_t, q_t\}$ with $p_t - q_t = p_s - q_s = p_k - q_k$. Furthermore, $t \neq s$ and $t \neq k$, since we have $p_k = p_t + w(t,s) + w(s,k)$. Proceeding in this way, we exhaust the set of vertices $v = 1, 2, \ldots, n-1$ with the result that one of the terms in the continued equality above must be $p_s - q_s = 0$. Hence, $p_v = q_v$ for all vertices.

Thus, if we terminated the above procedure when negative weights were present, we may end up with a set of values which may represent the length of walks instead of paths. Hence, arises the requirement for a sanity check. We shall prove that the same algorithm in this case, either outputs the lengths of the shortest paths or states if the graph has a negative cycle or not. Hence, we shall now create a list of edges L to see which ones cause improvement or not. We shall initialise as before $d(s)$ to 0 and $d(v)$ to $\infty \; \forall \; v \in V - S$. We then perform the procedure religiously $(n-1)$ times, in other words by using a dummy counter variable i, we do the following computation. In principle, what we are doing is, we are examining every edge and if it is improving then

```
1: for (i = 1 to n − 1)
2:         for (each e = (u, v) ∈ L)
3:                 if d(u) + w(u, v) < d(v)
4:                         d(v) = d(u) + w(u, v)
5:                         pred(v) = u
```

we are considering it else discarding it. An important thing, is that whenever we scan the list, every time it should be processed in the same direction, else we will be having an exponential solution. The time complexity is straightforward $O(nm)$, since we are performing $n-1$ passes and in each pass we are examining every edge which costs $O(m)$ time. Termination is also very

clear. As a sanity check, what we do, is to simply perform the procedure one more time i.e., make one more pass through L. If there is a negative cycle, then there is atleast one edge which misbehaves and alters the values in which case we shall output that the graph has a negative cycle. If however, they remain constant, then we shall show they are indeed the shortest path lengths only.

Before proceeding to the proof, let us try to see and understand what $d(v)$ in this case represents (just as it represented the lengths of the shortest special path in the case when all edge weigths were positive). To do so, we define a set $W(v, k)$ as follows

$$W(v, k) = \{\text{set of all walks from } s \text{ to } v \text{ with } \leq k \text{ edges}\}$$

It can be seen that k being a finite number, the set $W(v, k)$ is finite as well. Moreover, it should not be difficult to see that $W(v, k) \subseteq W(v, k + 1) \subseteq W(v, k + 2) \ldots$. We now say that $d_{\leq k}(v)$ is the length of the minimum walk in $W(v, k)$. Our claim is that, for every vertex, the following relation holds true.

$$d(v) \leq d_{\leq k}(v)$$

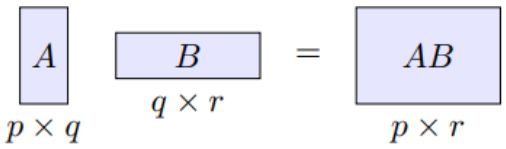
We prove this by induction with the base case being $k = 0$, which means basically that there is no edge from s to any vertex. In this case, the relation is true because $d(s) = 0$ and a walk from s to v without any edge is of inifinte length. Now, we assume its true for $(k - 1)$th pass and try to prove for k-th pass.

Using this result, we can see that after $(n - 1)$ iterations, we have $d(v) \leq d_{\leq(n-1)}(v)$. We see that if a graph has no negative cycles, then the length of the shortest walk is itself the length of the shortest path. To see why this is true, consider the case when the shortest walk has a positive cycle. Then we can remove all such edges and vertices constituting the positive cycle and obtain walk of shorter length and containing lesser number of vertices contradicting the fact that it was the shortest walk, hence it cant contain positive cycles. On the other hand, if it were to contain a zero cycle, then we can again remove it, in this case obtaining a walk with lesser number of edges.

12 Dynamic programming

§12.1 Case Study 1 : Matrix Multiplication

Given a matrix A with dimensions $p \times q$ and a matrix B with dimensions $q \times r$, then $C = AB$ is a $p \times r$ matrix, and calculating AB (naively) takes pqr multiplications as shown. Moreover in general, matrix multiplication is not commutative i.e., $AB \neq BA$. In fact, it doesn't even make sense to multiply matrices if their dimensions are not compatible. However, matrix multiplication is associative, which means that $(AB)C = A(BC)$.



The standard algorithm of multiplying two rectangular matrices is given by the procedure RECTANGULAR_MATRIX_MULTIPLY which computes $C = C + A.B$ for three matrices $A = (a_{ij})$, $B = (b_{ij})$ and $C = (c_{ij})$. Its running time is dominated by the number of scalar multiplications in line 4, which is pqr . Therefore, we'll consider the cost of multiplying matrices to be the number of scalar multiplications. (The number of scalar multiplications dominates even if we consider initializing $C = 0$ to perform just $C = A.B$)

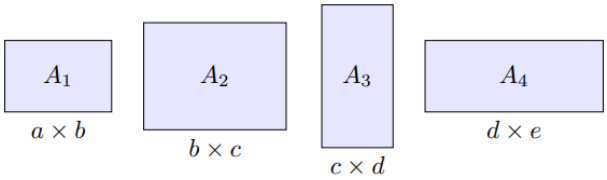
Algorithm RECTANGULAR_MATRIX_MULTIPLY (A, B, C, p, q, r)

```
1: for (i = 1 to p)
2:     for (j = 1 to r)
3:         for (k = 1 to q)
4:             c_ij = c_ij + a_ik.b_kj
```

Problem

Given a sequence (chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices to be multiplied, where the matrices aren't necessarily square, the goal is to compute the product $A_1 \times A_2 \times \cdots \times A_n$. using the standard algorithm for multiplying rectangular matrices, while minimizing the number of scalar multiplications. We can evaluate the product using the algorithm for multiplying pairs of rectangular matrices as a subroutine once we have parenthesized it to resolve all ambiguities in how the matrices are multiplied together. Matrix multiplication is associative, and so all parenthesizations yield the same product. A product of matrices is fully parenthesized if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses.

Remark 4 — If $\{A_i\}$ were all square $k \times k$ matrices, it wouldn't matter which order you multiply them in, since there would be $n-1$ matrix multiplications, each requiring k^3 scalar multiplications. The problem becomes interesting when these matrices are not all square matrices. We want to save operations by multiplying matrices in an efficient order.



For example, if the chain of matrices is $\langle A_1, A_2, A_3, A_4 \rangle$, then you can fully parenthesize the product $A_1A_2A_3A_4$ in five distinct ways :

$(A_1(A_2(A_3A_4)))$ takes $cde + bce + abe$ multiplications.
$(A_1((A_2A_3)A_4))$ takes $bcd + bde + abe$ multiplications.
$((A_1A_2)(A_3A_4))$ takes $abc + cde + ace$ multiplications.
$((A_1(A_2A_3))A_4)$ takes $bcd + abd + ade$ multiplications.
$(((A_1A_2)A_3)A_4)$ takes $abc + acd + ade$ multiplications.

Illustration 1 (*Different costs incurred by different parenthesizations of a matrix product*)

Consider the problem of a chain $\langle A_1, A_2, A_3 \rangle$ of three matrices. Suppose that the dimensions of the matrices are 10×100, 100×5 and 5×50, respectively. Multiplying according to the parenthesization $((A_1A_2)A_3)$ performs $10.100.5 = 5000$ scalar multiplications to compute the 10×5 matrix product A_1A_2, plus another $10.5.50 = 2500$ scalar multiplications to multiply this matrix by A_3, for a total of 7500 scalar multiplications. Multiplying according to the alternative parenthesization $(A_1(A_2A_3))$ performs $100.5.50 = 25,000$ scalar multiplications to compute the 100×50 matrix product A_2A_3 , plus another $10.100.50 = 50,000$ scalar multiplications to multiply A_1 by this matrix, for a total of 75,000 scalar multiplications. Thus, computing the product according to the first parenthesization is 10 times faster.

7 (***Matrix-Chain Multiplication Problem***)

Given a chain $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, where for $i = 1, 2, \ldots, n$ matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1A_2 \ldots A_n$ in a way that minimizes the number of scalar multiplications. The input is the sequence of dimensions $\langle p_0, p_1, p_2, \ldots, p_n \rangle$.

The matrix-chain multiplication problem does not entail actually multiplying matrices. The goal is only to determine an order for multiplying matrices that has the lowest cost. Typically, the time invested in determining this optimal order is more than paid for by the time saved later on when actually performing the matrix multiplications (such as performing only 7500 scalar multiplications instead of 75,000).

Counting the number of parenthesizations

Before solving the matrix-chain multiplication problem by dynamic programming, let us convince ourselves that exhaustively checking all possible parenthesizations is not an efficient algorithm. Denote the number of alternative parenthesizations of a sequence of n matrices by $P(n)$. When $n = 1$, the sequence consists of just one matrix, and therefore there is only one way to fully parenthesize the matrix product. When $n \geq 2$, a fully parenthesized matrix product is the product of two fully parenthesized matrix subproducts, and the split between the two subproducts may occur between the k-th and $(k+1)$-st matrices for any $k = 1, 2, \ldots, n-1$. Thus, we obtain the recurrence

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 \end{cases}$$

Aside 3 (*Catalan Numbers*)

Consider a similar recurrence relation

$$b_n = \begin{cases} 1 & \text{if } n = 0 \\ \sum_{k=0}^{n-1} b_k b_{n-1-k} & \text{if } n \geq 1 \end{cases}$$

Let $B(x)$ be the generating function,

$$B(x) = \sum_{n=0}^{\infty} b_n x^n$$

$$\begin{aligned} \implies B(x)^2 &= (b_0x^0 + b_1x^1 + b_2x^2 + \ldots)^2 \\ &= b_0^2x^0 + (b_0b_1 + b_1b_0)x^1 + (b_0b_2 + b_1b_1 + b_2b_0)x^2 + \ldots \\ &= \sum_{k=0}^{0} b_k b_{0-k} x^0 + \sum_{k=0}^{1} b_k b_{1-k} x^1 + \sum_{k=0}^{2} b_k b_{2-k} x^2 + \ldots \\ \therefore xB(x)^2 + 1 &= 1 + \sum_{k=0}^{0} b_k b_{1-1-k} x^0 + \sum_{k=0}^{1} b_k b_{2-1-k} x^1 + \sum_{k=0}^{2} b_k b_{3-1-k} x^2 + \ldots \\ &= 1 + b_1x^1 + b_2x^2 + b_3x^3 + \ldots \\ &= b_0x^0 + b_1x^1 + b_2x^2 + b_3x^3 + \ldots \\ &= \sum_{n=0}^{\infty} b_n x^n = B(x) \end{aligned}$$

Hence, one way to express $B(x)$ in closed form is

$$B(x) = \frac{1}{2x}(1 - \sqrt{1-4x})$$

$$\begin{aligned}\because xB(x)^2 + 1 &= x.\frac{1}{4x^2}(1 + 1 - 4x - 2\sqrt{1-4x}) + 1\\ &= \frac{1}{4x}(2 - 2\sqrt{1-4x}) - 1 + 1\\ &= \frac{1}{2x}(1 - \sqrt{1-4x}) = B(x)\end{aligned}$$

If we let $f(x) = \sqrt{1-4x}$, then the numerator of the derivative $f^{(k)}$ is

$$\begin{aligned}2.(1.2).(3.2).(5.2)\ldots &= 2^k.\prod_{i=0}^{k-2}(2k+1)\\ &= 2^k.\frac{(2(k-1))!}{2^{k-1}(k-1)!}\\ &= \frac{2(2(k-1))!}{(k-1)!}\end{aligned}$$

Moreover, $f(x) = 1 - 2x - 2x^2 - 4x^3 - 10x^4 - 28x^5 - \ldots$. Hence, we have

$$\begin{aligned}B(x) &= \frac{1}{2x}(1 - f(x))\\ &= 1 + x + 2x^2 + 5x^3 + 14x^4 + \ldots\\ &= \sum_{n=0}^{\infty}\frac{(2n)!}{(n+1)!n!}x\\ &= \sum_{n=0}^{\infty}\frac{1}{n+1}\frac{(2n)!}{n!n!}x\\ &= \sum_{n=0}^{\infty}\frac{1}{n+1}\binom{2n}{n}x\\ \implies b_n &= \frac{1}{n+1}\frac{(2n)!}{n!n!} && \text{which is the Catalan number}\\ &\approx \frac{1}{n+1}\frac{\sqrt{4\pi n}(2n/e)^{2n}}{2\pi n(n/e)^{2n}} && \text{from Stirling's approximation}\\ &= \frac{1}{n+1}\frac{4^n}{\sqrt{\pi n}}\\ &= \left(\frac{1}{n} + \left(\frac{1}{n+1} - \frac{1}{n}\right)\right)\frac{4^n}{\sqrt{\pi n}}\\ &= \left(\frac{1}{n} - \frac{1}{n^2+n}\right)\frac{4^n}{\sqrt{\pi n}}\\ &= \frac{1}{n}\left(1 - \frac{1}{n+1}\right)\frac{4^n}{\sqrt{\pi n}}\\ &= \frac{4^n}{\sqrt{\pi}n^{3/2}}(1 + O(1/n))\end{aligned}$$

Hence, we can see from the above discussion (aside 3) that the solution to a similar recurrence is the sequence of Catalan numbers, which grows as $\Omega(4^n/n^{3/2})$. Else, we can simply use the substitution method to show that the solution is $\Omega(2^n)$ as follows, suppose $P(n) \geq c2^n$,

$$\begin{aligned} P(n) &\geq \sum_{k=1}^{n-1} c2^k.c2^{n-k} \\ &= \sum_{k=1}^{n-1} c^2 2^n \\ &= c^2(n-1)2^n \\ &\geq c^2 2^n \qquad (n > 1) \\ &\geq c2^n \qquad (c \geq 1) \end{aligned}$$

The number of solutions is thus exponential in n, and the brute-force method of exhaustive search makes for a poor strategy when determining how to optimally parenthesize a matrix chain.

Applying dynamic programming

Let's use the dynamic-programming method to determine how to optimally parenthesize a matrix chain, by following the four step sequence :

Step 1 : *Characterize the structure of an optimal solution*

We find the optimal sub-structure and then use it to construct an optimal solution to the problem from optimal solutions to subproblems.

- An optimal ordering of the product $A_1A_2 \ldots A_n$ splits the product between A_k and A_{k+1} for some k :
$$A_2 \ldots A_n = \underline{A_1 \ldots A_k} \,.\, \underline{A_{k+1} \ldots A_n}$$
- Key observation : The ordering of $A_1 \ldots A_k$ within this ("global") optimal ordering must be an optimal ordering of (sub-product) $A_1 \ldots A_k$ [1]
- Similar observation holds for $A_{k+1} \ldots A_n$
- Thus, an optimal ("global") solution contains within it the optimal ("local") solutions to subproblems. **(the optimal substructure property)**

[1] Why? simply argue by contradiction : If there were a less costly way to order the product $A_1 \ldots A_k$, substituting that ordering within this (global) optimal ordering would produce another ordering of $A_1A_2 \ldots A_n$, whose cost would be less than the optimum, a contradiction!

Step 2 : *Recursively define the value of an optimal solution*

- Define $m[i, j]$ as the minimum number of scalar multiplications needed to compute $A_i \dots A_j$
- $m[1, n]$ is then by definition, the cheapest way for the product $A_1 A_2 \dots A_n$
- $m[i, j]$ equals the minimum cost $m[i, k]$ for computing the subproduct $A_{i:k}$, plus the minimum cost $m[k + 1, j]$ for computing the subproduct, $A_{k+1:j}$, plus the cost of multiplying these two matrices together. Since each matrix A_i is $p_{i-1} \times p_i$, computing the matrix product $A_{i:k} A_{k+1:j}$ takes $p_{i-1} p_k p_j$ scalar multiplications
- $m[i, j]$ can be defined recursively as follows for $1 \le 1 \le j \le n$,

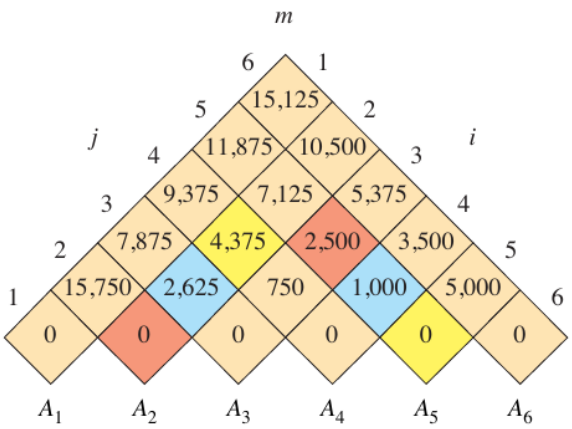
$$m[i, j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k \le j} \{m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j\} & \text{if } i < j \end{cases}$$

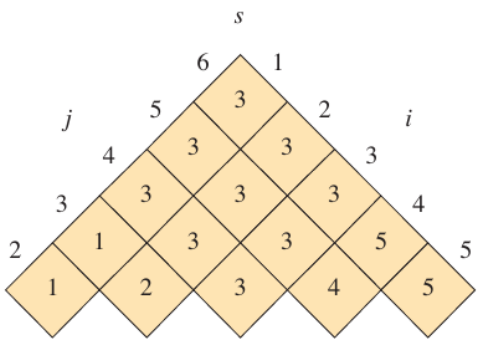
- To construct an optimal ordering, we define $s[i, j]$ to be a value of k at which you split the product $A_i A_{i+1} \dots A_j$ in an optimal parenthesization i.e., $s[i, j]$ equals a value k such that $m[i, j] = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$

Let us see an example so as to how the m and s tables computed by this recurrence appear.

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30 x 35	35 x 15	15 x 5	5 x 10	10 x 20	20 x 25

The tables are rotated so that the main diagonal runs horizontally. The m table uses only the main diagonal and upper triangle, and the s table uses only the upper triangle. The minimum number of scalar multiplications to multiply the 6 matrices is $m[1, 6] = 15,125$. Of the entries that are not tan, the pairs that have the same color are taken together when computing the recurrence.





As an example, lets compute and see $m[2,5]$

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35.15.20 = 13,000 \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35.5.20 = 7125 \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35.10.20 = 11,375 \end{cases}$$

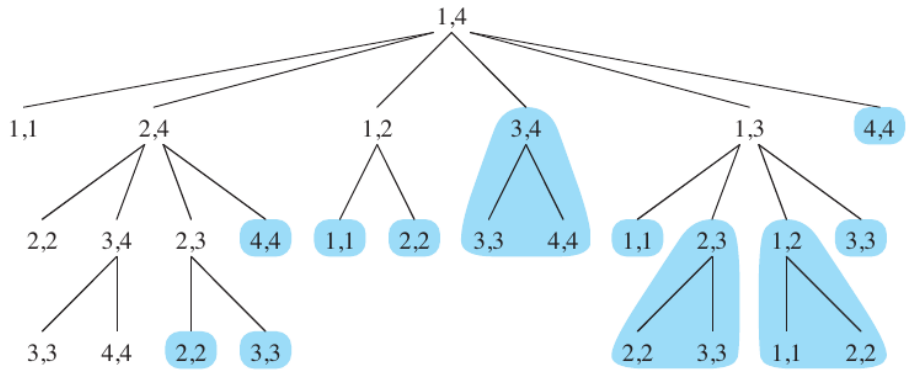
When a recursive algorithm revisits the same problem repeatedly, we say that the optimization problem has ***overlapping subproblems***. For example, consider the inefficient (we shall show why so soon) recursive procedure RECURSIVE_MATRIX_CHAIN which determines $m[i, j]$. The procedure is based directly on the above recurrence relation.

Algorithm RECURSIVE_MATRIX_CHAIN(p, i, j)

```
1: if (i = j)
2:         return 0
3: m[i, j] = ∞
4: for (k = i to j − 1)
5:         q = RECURSIVE_MATRIX_CHAIN(p, i, k)
6:                 + RECURSIVE_MATRIX_CHAIN(p, k + 1, j) + p_{i−1} p_k p_j
7:         if (q < m[i, j])
8:                 m[i, j] = q
9: return m[i, j]
```

Below figure shows the recursion tree produced by the call RECURSIVE_MATRIX_CHAIN(p, 1, 4). Each node is labeled by the values of the parameters i and j. Observe that some pairs of values occur many times. In fact, the time to compute $m[1, n]$ by this recursive procedure is at least exponential in n. To see why, let $T(n)$ denote the time taken by RECURSIVE_MATRIX_CHAIN to compute an optimal parenthesization of a chain of n matrices. Because the execution of lines 1-2 and of lines 6-7 each take at least unit time, as does the multiplication in line 5, inspection of the procedure yields the following recurrence.

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



Noting that for $i = 1, 2, \ldots, n-1$, each term $T(i)$ appears once as $T(k)$ and once as $T(n-k)$, and collecting the $n-1$ 1s in the summation together with the 1 out front, we can rewrite the recurrence as

$$T(n) \geq 2 \sum_{i=1}^{n-1} T(i) + n$$

Let's prove that $T(n) = \Omega(2^n)$ using the substitution method. Specifically, we'll show that $T(n) \geq 2^{n-1}$ for all $n \geq 1$. For the base case $n = 1$, the summation is empty, and we get $T(1) \geq 1 = 2^0$. Inductively, for $n \geq 2$ we have

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

which completes the proof. Thus, the total amount of work performed by the call RECURSIVE_MATRIX_CHAIN $(p, 1, n)$ is at least exponential in n. As a practical matter, you'll often want to store in a separate table which choice you made in each subproblem so that you do not have to reconstruct this information from the table of costs. The idea is to ***memoize*** the natural, but inefficient, recursive algorithm which offers the efficiency of the bottom-up approach (that is using the optimal substructure property, we first find optimal solutions to subproblems and, having solved the subproblems, we find an optimal solution to the problem) while maintaining a top-down strategy. As in the bottom-up approach, you maintain a table with subproblem solutions, but the control structure for filling in the table is more like the recursive algorithm.

A memoized recursive algorithm maintains an entry in a table for the solution to each subproblem. Each table entry initially contains a special value to indicate that the entry has yet to be filled in. When the subproblem is first encountered as the recursive algorithm unfolds, its solution is computed and then stored in the table. Each subsequent encounter of this subproblem simply looks up the value stored in the table and returns it. (This approach presupposes that you know the set of all possible subproblem parameters and that you have established the relationship between table positions and subproblems. Another, more general, approach is to memoize by using hashing with the subproblem parameters as keys). The memoized procedure MEMOIZED_MATRIX_CHAIN runs in $O(n^3)$ time. To begin with, line 4 of MEMOIZED_MATRIX_CHAIN executes $\Theta(n^2)$ times, which dominates the running time outside of the call to LOOKUP_CHAIN in line 5. We can categorize the calls of LOOKUP_CHAIN into two types :

1. calls in which $m[i, j] = \infty$, so that lines 3-12 execute
2. calls in which $m[i, j] < \infty$, so that LOOKUP_CHAIN simply returns in line 2

There are $\Theta(n^2)$ calls of the first type, one per table entry. All of the second type are made as recursive calls of the first type. Whenever a given call of LOOKUP_CHAIN makes recursive calls, it makes $O(n)$ of them. Hence, there are $O(n^3)$ calls of the second type in all. Each call of the second type takes $O(1)$ time and each call of the first type takes $O(n)$ time plus the time spent in its recursive calls. The total time is therefore $O(n^3)$. Memoization thus turns an $\Omega(2^n)$-time algorithm into an $O(n^3)$-time algorithm.

Step 3 : *Compute the value of an optimal solution*

Algorithm MEMOIZED_MATRIX_CHAIN(p, n)

```
1: let m[1 : n, 1 : n] and s[1 : n − 1, 2 : n] be new tables
2: for (i = 1 to n)
3:         for (j = i to n)
4:                 m[i, j] = ∞
5: return LOOKUP_CHAIN(m, p, 1, n)
```

Algorithm LOOKUP_CHAIN$(m, p, 1, n)$

```
1: if (m[i, j] < ∞)
2:         return m[i, j]
3: if (i = j)
4:         m[i, j] = 0
5: else
6:         for (k = i to j − 1)
7:                 q = LOOKUP_CHAIN(m, p, i, k)
8:                         + LOOKUP_CHAIN(m, p, k + 1, j) + p_{i−1} p_k p_j
9:                 if (q < m[i, j])
10:                        m[i, j] = q
11:                        s[i, j] = k
12: return m[i, j]
```

Each entry $s[i, j]$ records a value of k such that an optimal parenthesization of $A_i A_{i+1} \dots A_j$ splits the product between A_k and A_{k+1}. The final matrix multiplication in computing $A_{1:n}$ optimally is $A_{1:s[1,n]} A_{s[1,n]+1:n}$. The s table contains the information needed to determine the earlier matrix multiplications as well, using recursion : $s[1, s[1, n]]$ determines the last matrix multiplication when computing $A_{1:s[1,n]}$ and $s[s[1, n] + 1, n]$ determines the last matrix multiplication when computing $A_{s[1,n]+1:n}$. The initial call PRINT_OPTIMAL_PARENS$(s, 1, n)$ prints an optimal parenthesization of the full matrix chain product $A_1 A_2 \dots A_n$. Thus, we have examined the two key ingredients that an optimization problem must have in order for dynamic programming to apply : ***optimal substructure*** and ***overlapping subproblems***. We have also seen how memoization might help us take advantage of the overlapping-subproblems property in a top-down recursive approach.

Step 4 : *Construct an optimal solution from computed information*

Algorithm PRINT_OPTIMAL_PARENS(s, i, j)

```
1: if (i = j)
2: print "A_i"
3: else
4:         print "("
5:         PRINT_OPTIMAL_PARENS(s, i, s[i, j])
6:         PRINT_OPTIMAL_PARENS(s, s[i, j] + 1, j)
7:         print ")"
```

An iterative version of step 3, using bottom down approach is as follows :

Algorithm MATRIX_CHAIN_ORDER(p, n)

```
1: let m[1 : n, 1 : n] and s[1 : n − 1, 2 : n] be new tables
2: for (i = 1 to n)
3:         m[i, i] = 0
4: for (l = 2 to n)
5:         for (i = 1 to n − l + 1)
6:                 j = i + l − 1
7:                 m[i, j] = ∞
8:                 for (k = 1 to j − 1)
9:                         q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10:                        if (q < m[i, j])
11:                                m[i, j] = q
12:                                s[i, j] = k
13: return m and s
```

To summarize, the dynamic-programming method works as follows. Instead of solving the same subproblems repeatedly, as in the naive recursion solution, arrange for each subproblem to be solved only once. There's actually an obvious way to do so : the first time you solve a subproblem, save its solution. If you need to refer to this subproblem's solution again later, just look it up, rather than recomputing it. Saving subproblem solutions comes with a cost : the additional memory needed to store solutions. Dynamic programming thus serves as an example of a ***time - memory trade - off***. There are usually two equivalent ways to implement a dynamic-programming approach.

- ***Top-down with memoization*** : In this approach, you write the procedure recursively in a natural manner, but modified to save the result of each subproblem (usually in an array or hash table). The procedure now first checks to see whether it has previously solved this subproblem. If so, it returns the saved value, saving further computation at this level. If not, the procedure computes the value in the usual manner but also saves it. We say that

the recursive procedure has been memoized: it "remembers" what results it has computed previously.

- ***Bottom-up method*** : This approach typically depends on some natural notion of the "size" of a subproblem, such that solving any particular subproblem depends only on solving "smaller" subproblems. Solve the subproblems in size order, smallest first, storing the solution to each subproblem when it is first solved. In this way, when solving a particular subproblem, there are already saved solutions for all of the smaller subproblems its solution depends upon. You need to solve each subproblem only once, and when you first see it, you have already solved all of its prerequisite subproblems.

In general practice, if all subproblems must be solved at least once, a bottom-up dynamic-programming algorithm usually outperforms the corresponding top-down memoized algorithm by a constant factor, because the bottom-up algorithm has no overhead for recursion and less overhead for maintaining the table. Moreover, for some problems you can exploit the regular pattern of table accesses in the dynamic programming algorithm to reduce time or space requirements even further. On the other hand, in certain situations, some of the subproblems in the subproblem space might not need to be solved at all. In that case, the memoized solution has the advantage of solving only those subproblems that are definitely required.

§12.2 Case Study 2 : Longest Common Subsequence

A subsequence of a given sequence is just the given sequence with 0 or more elements left out. Formally, given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, another sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a ***subsequence*** of X if there exists a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of X such that for all $j = 1, 2, \ldots, k$ we have $x_{i_j} = z_j$. We defined the i-th ***prefix*** of X, for $i = 0, 1, \ldots, m$ as $X_i = \langle x_1, x_2, \ldots, x_i \rangle$. Given two sequences X and Y, we say that a sequence Z is a ***common subsequence*** of X and Y if Z is a subsequence of both X and Y.

Example : If $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$ the sequence $\langle B, C, A \rangle$ is a common subsequence of both X and Y . The sequence $\langle B, C, A \rangle$ is not a ***longest common subsequence (LCS)*** of X and Y, however, since it has length 3 and the sequence $\langle B, C, B, A \rangle$ which is also common to both sequences X and Y, has length 4. The sequence $\langle B, C, B, A \rangle$ is an LCS of X and Y, as is the sequence $\langle B, D, A, B \rangle$ since X and Y have no common subsequence of length 5 or greater. Moreover, $X_4 = \langle A, B, C, B \rangle$ and X_0 is the empty sequence.

8 (Longest-Common-Subsequence Problem)
Given two input sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$, our goal is to find a maximum length common subsequence of X and Y

We can solve the LCS problem with a brute-force approach: enumerate all subsequences of X and check each subsequence to see whether it is also a subsequence of Y, keeping track of the longest subsequence you find. Each subsequence of X corresponds to a subset of the indices

$\{1, 2, \ldots, m\}$ of X. Since X has 2^m subsequences, this approach requires exponential time, making it impractical for long sequences. We shall now see how to efficiently solve the LCS problem using dynamic programming.

Theorem 2 (*Optimal substructure of an LCS*)

Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y

- If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}
- If $x_m \neq y_n$ and $z_k \neq x_m$, then Z is an LCS of X_{m-1} and Y
- If $x_m \neq y_n$ and $z_k \neq y_n$, then Z is an LCS of X and Y_{n-1}

Proof : (1) If $z_k \neq x_m$ then we could append $x_m = y_n$ to Z to obtain a common subsequence of X and Y of length $k + 1$, contradicting the supposition that Z is a longest common subsequence of X and Y. Thus, we must have $z_k = x_m = y_n$. Now, the prefix Z_{k-1} is a length - $(k - 1)$ common subsequence of X_{m-1} and Y_{n-1}. We wish to show that it is an LCS. Suppose for the purpose of contradiction that there exists a common subsequence W of X_{m-1} and Y_{n-1} with length greater than $k - 1$. Then, appending $x_m = y_n$ to W produces a common subsequence of X and Y whose length is greater than k, which is a contradiction.

(2) If $z_k \neq x_m$, then Z is a common subsequence of X_{m-1} and Y. If there were a common subsequence W of X_{m-1} and Y with length greater than k, then W would also be a common subsequence of X_m and Y, contradicting the assumption that Z is an LCS of X and Y.

(3) The proof is symmetric to (2).

Step 1 : *Characterize the structure of an optimal solution*

Let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$. Then,

- Case 1. If $x_m = y_n$ then
  - (a) $z_k = x_m = y_n$
  - (b) $Z_{k-1} = \langle z_1, z_2, \ldots, z_{k-1} \rangle = \text{LCS}(X_{m-1}, Y_{n-1})$
- Case 2. If $x_m \neq y_n$ then
  - (a) $z_k \neq x_m \implies Z_k = \text{LCS}(X_{m-1}, Y_n)$
  - (b) $z_k \neq y_n \implies Z_k = \text{LCS}(X_m, Y_{n-1})$

In other words, the optimal solution to the (whole) problem contains within it the optimal solutions to subproblems

Step 2 : *Recursively define the value of an optimal solution*

- Define $c[i, j]$ as the length of LCS(X_i, Y_j)
- $c[m, n]$ is then by definition the length of LCS(X_m, Y_n)
- If $x_m = y_n$ we need to find an LCS of X_{m-1} and Y_{n-1}. Appending $x_m = y_n$ to this LCS yields an LCS of X and Y. If $x_m \neq y_n$, then we have to solve two subproblems : finding an LCS of X_{m-1} and Y and finding an LCS of X and Y_{n-1}. Whichever of these two LCSs is longer is an LCS of X and Y.
- $c[i, j]$ is defined recursively as follows for $0 \leq i \leq m$ and $0 \leq j \leq n$:

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max\{c[i, j-1], c[i-1, j]\} & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

The procedure LCS_LENGTH also maintains the table $b[1 : m, 1 : n]$ to help in constructing an optimal solution. Intuitively, $b[i, j]$ points to the table entry corresponding to the optimal subproblem solution chosen when computing $c[i, j]$.

Step 3 : *Compute the value of an optimal solution*

Algorithm LCS_LENGTH(X, Y, m, n)

```
1:  let b[1 : m, 1 : n] and c[0 : m, 0 : n] be new tables
2:  for (i = 1 to m)
3:          c[i, 0] = 0
4:  for (j = 0 to n)
5:          c[0, j] = 0
6:  for (i = 1 to m)
7:          for (j = 1 to n)
8:                  if (x_i = y_j)
9:                          c[i, j] = c[i − 1, j − 1] + 1
10:                         b[i, j] = " ↖ "
11:                 elif (c[i − 1, j] ≥ c[i, j − 1])
12:                         c[i, j] = c[i − 1, j]
13:                         b[i, j] = " ↑ "
14:                 else
15:                         c[i, j] = c[i, j − 1]
16:                         b[i, j] = " ← "
17: return c and b
```

The LCS problem has only $\Theta(mn)$ distinct subproblems (computing $c[i, j]$ for $0 \leq i \leq m$ and

$0 \le j \le n$), hence its running time is $\Theta(mn)$, since each table entry takes $\Theta(1)$ time to compute. (Note that we compute solutions bottom-up) The initial call, PRINT_LCS(b, X, m, n) gives the elements of LCS in proper, forward error. The procedure takes $O(m+n)$ time, since it decrements at least one of i and j in each recursive call.

Step 4 : *Construct an optimal solution from computed information*

Algorithm PRINT_LCS(b, X, i, j)

```
if (i = 0 or j = 0)
return
if (b[i, j] = " ↖ ")
        PRINT_LCS(b, X, i − 1, j − 1)
        print x_i
elif (b[i, j] = " ↑ ")
        PRINT_LCS(b, X, i − 1, j)
else
        PRINT_LCS(b, X, i, j − 1)
```

The c and b tables computed by LCS_LENGTH on the sequences $X = \langle A, B, C, B, D, A, B\rangle$ and $Y = \langle B, D, C, A, B, A\rangle$. The entry 4 in $c[7, 6]$ - the lower right-hand corner of the table - is the length of an LCS $\langle B, C, B, A\rangle$ of X and Y. To reconstruct the elements of an LCS, follow the $b[i, j]$ arrows from the lower right-hand corner, as shown by the sequence shaded blue. Each ↖ on the shaded-blue sequence corresponds to an entry (highlighted) for which $x_i = y_j$ is a member of an LCS.

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

IV

Practice Problems

13 Decremental Design

Problem 13A. Binary GCD algorithm : Most computers can perform the operations of subtraction, testing the parity (odd or even) of a binary integer, and halving more quickly than computing remainders. This problem investigates the binary gcd algorithm, which avoids the remainder computations used in Euclid's algorithm.

(a) Prove that if a and b are both even, then $\gcd(a, b) = 2 \cdot \gcd(a/2, b/2)$.

(b) Prove that if a is odd and b is even, then $\gcd(a, b) = \gcd(a, b/2)$.

(c) Prove that if a and b are both odd, then $\gcd(a, b) = \gcd((a - b)/2, b)$.

(d) Design an efficient binary gcd algorithm for input integers a and b, where $a \geq b$, that runs in $O(\lg a)$ time. Assume that each subtraction, parity test, and halving takes unit time. [$\lg a$ is the same as $\log_2 a$.]

Solution 13.1 — (a) Let $a = 2k_1$ and $b = 2k_2$ and $d = \gcd(a, b)$. Since, d divides a and b both, and also both are even numbers,therefore d must be even too. Now, d divides a, and both have a common factor of 2, So $d/2$ must divide $a/2$. Similarly for $b/2$. Since $a/2$ and $b/2$ contains a common factor $d/2$, $\gcd(a/2, b/2) = d/2$. Hence, $\gcd(a, b) = 2 \cdot \gcd(a/2, b/2)$.

(b) Let $a = 2k_1 + 1$ and $b = 2k_2$ and $d = \gcd(a, b)$. Since, d divides a, and a is an odd number, therefore d must be odd. Now, b is an even number and d divides b and d is odd, this implies that d must divide $b/2$ as d doesn't contain any factor of 2. Therefore, a and $b/2$ has a greatest common factor d. Therefore, $\gcd(a, b) = \gcd(a, b/2)$.

(c) Let $a = 2k_1 + 1$ and $b = 2k_2 + 1$ and $d = \gcd(a, b)$. Using Euclid's Theorem, we can write $\gcd(a, b) = \gcd(a - b, b)$. Now, let $a - b = c$. So, $c = 2.(k_1 - k_2)$, which means that c is even. Hence, using the above case, we can write $\gcd(a - b, b) = \gcd((a - b)/2, b)$. Therefore, $\gcd(a, b) = \gcd((a - b)/2, b)$.

(d)Here, is the algorithm implementation in C++

```
int gcd (int a , int b )
{
    if (( a ==1) || ( b ==1) ) return 1;
    if( !( a %2) )
        if( !( b %2) ) return ( gcd ( a >> 1 , b >> 1) << 1) ;
        else return gcd ( a >> 1 , b ) ;
```

```
    else
        if( !( b %2) ) return gcd (a , b >> 1) ;
        else return gcd (( a - b ) >> 1 , b ) ;
}
```

Problem 13B. Suppose you are given an array A with n entries, with each entry holding a distinct number. You are told that the sequence of values $A[1], A[2], \ldots, A[n]$ is unimodal : For some index p between 1 and n, the values in the array entries increase up to position p in A and then decrease the remainder of the way until position n. (So if you were to draw a plot with the array position j on the x-axis and the value of the entry $A[j]$ on the y-axis, the plotted points would rise until x-value p, where they'd achieve their maximum, and then fall from there on.) You'd like to find the "peak entry" p without having to read the entire array-in fact, by reading as few entries of A as possible. Show how to find the entry p by reading at most $O(\log n)$ entries of A.

Solution 13.2 — Central Idea : Here, as we know that there must exist one peak, and also elements will follow an increasing order till the peak and decreasing order after that, So, we can access middle element of each region and then by comparing the gradient using the previous and next value of that element, we can find the direction of increment or decrement. If, in both directions, the value decreases, it implies that we have found the peak. We can return that location. So, we will look at the gradient at the mid location; accordingly, we will reduce the range to half at each step by finding the direction of increment each time. Finally, if we find that elements are decreasing on both sides, we will stop. Here, is the implementation in C++,

```
int peakfind (int arr [] , int N )
{
    int init = ( N /2) ;
    while (1)
    {
        if( arr [ init ] < arr [ init + 1]) init += ( init /2) ;
        else
            if( arr [ init ] < arr [ init -1]) init -= ( init /2) ;
            else return init ;
    }
}
```

Proof of Correctness

Because of the given constraints, there will never be a case when elements are increasing on both sides. So, first, we go at the middle element of the given range, we check the next element of it, if it is higher than the middle element, it means that peak must be present in the right half part. So, we increase our *init* by *init*/2. Similarly, we check for the element previous to the central element; if it is higher than the middle element, it means that peak should be in the left half part, So, we decrease our *init* by *init*/2. If both sides are decreasing,

we found the peak, and so we return it.

Time Complexity Analysis

As explained in the text above, we are reducing the range in half at each step, So, we will take at most log(N) steps, where N = size of the array provided. And, in every step we will access two locations, which is a constant amount of work therefore, total time complexity of the algorithm is $O(\log N)$.

Problem 13C. Consider an n-node complete binary tree T, where $n = 2^d - 1$ for some d. Each node v of T is labelled with a real number x_v. You may assume that the real numbers labelling the nodes are all distinct. A node v of T is a local minimum if the label x_v is less than the label x_w for all nodes w that are joined to v by an edge. You are given such a complete binary tree T, but the labelling is only specified in the following implicit way: for each node v, you can determine the value x_v by probing the node v. Show how to find a local minimum of T using only $O(\log n)$ probes to the nodes of T.

Solution 13.3 — **Note :** We are using a fact that if a leaf is smaller than it's parent, then it is a local minimum.

Central Idea : To get the local minimum in $O(\log N)$, we will divide the tree recursively and then search it. Here, is the implementation in C++.

```
class Node
{
public :
    static int mindata ;
    int data ;
    Node * left ,* right ;
    Node ( int x )
    {
        data = x ;
        left = right = nullptr ;
    }
};
int Node :: mindata ;
void recur ( Node * head )
{
    if( head - > left != nullptr )
    {
        if( head - > data > head - > left - > data ) recur ( head - >
        ↪  left ) ;
```

```
19          else if( head -> data > head -> right -> data ) recur (
            ↪  head -> right ) ;
20          else Node :: mindata = head -> data ;
21      }
22      else Node :: mindata = head -> data ;
23  }
```

Proof of Correctness

Claim 1 : There must exist a local minimum in a tree with the given constraints.
Proof : Any finite set with a total order has a minimum. Since the binary has finite number of elements, we can say that there must exist a global minimum for the tree and as a global minimum is also a local minimum, a local minimum will always exist.

Claim 2 : If one child is smaller than it's parent, then local minimum must exist in it's subtree.
Proof : Assume that the claim is not true. Then we can construct a case such that both the children of the root are smaller than the root and no local minimum exists in both of their subtrees. And as root is greater than both of its children, it is also not a local minimum. Therefore, no local minimum exists in the tree, which contradicts claim 1, proved above. So, claim 2 is true.

Claim 3 : The given algorithm will terminate.
Proof : If we start from the root, if the algorithm is non-terminating, it will surely reach a leaf in it's path as we are recursively searching left and right children of the root. Considering the fact that if a node is a leaf and it is smaller than it's parent, we have found our local maximum. Also, if a node has two children both of which are leaves, and both are greater than it's value , that node is the local minimum. Hence proved, this algorithm will terminate.

Using the above claims, we have proved that the above algorithm will terminate and will return the local minimum

Time Complexity Analysis

According to the algorithm shown above, we start from the root; if it's value is less than it's left child, it will recursively check it's left subtree and it is bound to get a local minimum as proved above. Which means that right subtree will never be accessed. If, value of left child is greater than the root, it will recursively search right subtree, where again it is bound to get a local minima. So, we will never access the left subtree and if both of it's child are greater than , root is the local minima and so we will not access any of it's subtree. So, at every step, we are reducing our search space by half, which means total no. of accesses as well as the time complexity will be $O(\log N)$.

Problem 13D. Suppose now that you're given an $n \times n$ grid graph G. (An $n \times n$ grid graph is just

the adjacency graph of an $n \times n$ chessboard. To be completely precise, it is a graph whose node set is the set of all ordered pairs of natural numbers (i, j), where $1 \leq i \leq n$ and $1 \leq j \leq n$; the nodes (i, j) and (k, l) are joined by an edge if and only if $|i - k| + |j - l| = 1$. We use some of the terminology of the previous question. Again, each node v is labeled by a real number x_v; you may assume that all these labels are distinct. Show how to find a local maximum of G using only $O(n)$ probes to the nodes of G. (Note that G has n^2 nodes.)

Solution 13.4 — Lets see the solution step by step.

Algorithm explained

1. Take the "window frame" formed by the first, middle, and last row, and first, middle, and last column. Find the maximum element of these $6n - 9$ elements, $m = a[i][j]$.
2. If the maximum element is greater than all of its neighbours we are done, we return the respective indices i and j.
3. Otherwise there exists a neighbour of m which is greater than it and does not lie on the window (as m is the maximum element on window). We use recursion in this window.

Assumptions

1. The values of $a[0][i], a[i][0], a[n+1][i]$ and $a[i][n+1]$ are zero
2. The matrix values are 1-indexed and therefore are stored from $a[1][1]$ to $a[n][n]$. 3. The functions is called with the following paramaters findPeak$(1, n, 1, n)$

Here is the implementation in C++,

```
pair <int , int > findPeak ( int xl , int xr , int yl , int yr )
{
    // mx stores position of middle column and my stores position of
     ↪  middle row .
    int mx = xl +( xr - xl ) /2;
    int my = yl +( yr - yl ) /2;
    // maxx stores the maximum element in the window and xi ,yi
     ↪  store its index
    int max = INT_MIN , xi , yi ;
    for ( int i = xl ; i <= xr ; i ++)
        if( maxx < a [ i ][ my ])
            maxx = a [ i ][ my ] , xi =i , yi = my ;
        else if( maxx < a [ i ][ yl ])
            maxx = a [ i ][ yl ] , xi =i , yi = yl ;
        else if( maxx < a [ i ][ yr ])
            maxx = a [ i ][ yr ] , xi =i , yi = yr ;
    for ( int i = yl ; i <= yr ; i ++)
```

```
16          if( maxx < a [ mx ][ i ])
17              maxx = a [ mx ][ i ] , xi = mx , yi = i ;
18          else if( maxx < a [ xl ][ i ])
19              maxx = a [ xl ][ i ] , xi = xl , yi = i ;
20          else if( maxx < a [ xr ][ i ])
21              maxx = a [ xr ][ i ] , xi = xr , yi = i ;
22      if( maxx > a [ xi -1][ yi ] && maxx > a [ xi ][ yi -1] && maxx >
        ↪  a [ xi ][ yi +1] && maxx > a [ xi +1][ yi ])
23          return ({ xi , yi }) ;
24      else
25      {
26          if( xi == xl )
27              return findPeak ( xi +1 , mx -1 ,( yi > my ) ? my +1: yl
                ↪  ,( yi > my ) ? yr : my -1) ;
28          else if( xi == xr )
29              return findPeak ( mx +1 , xi -1 ,( yi > my ) ? my +1: yl
                ↪  ,( yi > my ) ? yr : my -1) ;
30          else if( xi == mx )
31              if( maxx < a [ xi +1][ yi ])
32                  return findPeak ( xi +1 , mx -1 ,( yi > my ) ? my +1:
                    ↪  yl ,( yi > my ) ? yr : my -1) ;
33              else
34                  return findPeak ( mx +1 , xi -1 ,( yi > my ) ? my +1:
                    ↪  yl ,( yi > my ) ? yr : my -1) ;
35          if( yi == yl )
36              return findPeak (( xi > mx ) ? mx +1: xl ,( xi > mx ) ?
                ↪  xr : mx -1 , yi +1 , my -1) ;
37          else if( yi == yr )
38              return findPeak (( xi > mx ) ? mx +1: xl ,( xi > mx ) ?
                ↪  xr : mx -1 , my +1 , yi -1) ;
39          else if( yi == my )
40              if( maxx < a [ xi ][1+ yi ])
41                  return findPeak (( xi > mx ) ? mx +1: xl ,( xi > mx )
                    ↪  ? xr : mx -1 , my +1 , yi -1) ;
42              else
43                  return findPeak (( xi > mx ) ? mx +1: xl ,( xi > mx )
                    ↪  ? xr : mx -1 , yi +1 , my -1) ;
44      }
45  }
```

Proof of Correctness

Claim : If you recurse on a quadrant, there is indeed a peak in that quadrant

Proof : The quadrant we selected contains an element larger than m. Thus we know that the maximum element in this quadrant must also be larger than m. Since m is the maximum element surrounding this quadrant, the maximum element in this quadrant must be larger than any element surrounding this quadrant. This element must be greater than or equal to all of its neighbors since it is greater than all elements within the quadrant and directly outside of the quadrant.

Time Complexity Analysis

From the above Program we can see that,

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

and from base case we know that

$$T(n, 1) = T(1, n) = kn$$

The proof follows,

$$\begin{aligned} T(n, n) &= T(n/2, n/2) + 2kn \\ T(n/2, n/2) &= T(n/4, n/4) + kn \\ T(n/4, n/4) &= T(n/8, n/8) + kn/2 \\ &\vdots \end{aligned}$$

We also know that, $2n + n + n/2 + \cdots = 4n$, hence summing up the above series we get $T(n, n) \leq 4kn$. Thus, $T(n)$ is $O(n)$ and hence proved.

Problem 13E. Suppose $S = 1, 2, \ldots, n$ and $f : S \to S$. If $R \subset S$, define $f(R) = \{f(x) \mid x \in R\}$. Device an $O(n)$ algorithm for determining the largest $R \subset S$, such that $f(R) = R$.

Solution 13.5 — Lets see the solution step by step.

Algorithm explained

First of all, start with an array *arr* which is initialized to -1 for all integers n. Next, create a queue q which will hold the current cycle we will check. To keep a track of cycles, we maintain a variable t. Now, start from first element and go on traversing the list till an index is found such that $arr[x] = -1$ or to say it is not yet considered. Insert it in the queue q. While the queue is not empty, pop one element from it and let it be x. Now, if $arr[x] = -1$, it means that it is not yet considered , so mark it $-t$. If $arr[x] = -t$ for the current cycle, then push it in a vector *pos* and mark it with t. If it is not in any of this then set a $flag = false$, which means that the currently present element in *pos* do not form a cycle. After that, do the same thing for $f(x)$ and if it is -1 then push it in the queue. If it is $-t$, push it in the *pos* and mark it with t. After the queue becomes empty, if flag is not deasserted, the elements in the

pos makes a cycle and so mark all of them as t. After the first step, all the elements, which have $arr[x] < 0$, should not be in the set. So, iterate the complete *arr* and find all such set of cycles, or $arr[x]$ with $arr[x] > 0$. If, the size of this set is $< S$, then return this as R, else, remove the minimum cycle from the set and then return it as R.

Proof of Correctness

Claim : If and only if R is a set of elements such that it forms a set of cycles within it, then $f(R) = R$
Proof : We shall see the forward direction first. Let there be n elements in the cycle namely $x_1, x_2, \ldots, x_n$. As they form a cycle, therefore, $f(x_1) = x_2, f(x_2) = x_3, \ldots, f(x_{(n-1)}) = x_n$ and $f(x_n) = x_1$. As seen each and every element is in R, therefore the forward direction holds. Now, the backward direction. Given $f(R) = R$, proving with induction on length,

- **Base case :** length = 1, only one possibility $f(x) = x$, so given claim holds true
- **Induction step :** let the given claim hold true for some length k.
- **Induction hypothesis :** For length $k + 1$, we need to insert one more element in domain as well as range to maintain $f(R) = R$. There can be two cases,
  1. $f(x) = x$, then it forms a self loop and so , the claim holds true.
  2. $f(x) = y$, where $y \in R'$, this way we have inserted x in the domain of R, but still to maintain $f(R) = R$, we need to insert it in the range also. Assume that for some a, $f(a) = x$,then we have inserted x in range but, the old value of $f(a)$ will be removed out of the range as we broke the link. So, if we want to maintain $f(R) = R$, we need to assure that $y = f(a)$, then again x will be a part of the cycle containing a and y and so the claim holds true.

Hence proved , the given claim holds true.

As seen in the above claim, for $f(R) = R$ to be true, R should contain only cycles and self loops. Now, our task is to find all the set of cycles in the given set S. We find that by maintaining an *arr*, which hold the status of that element, we initialize it with -1, and if it is found in domain or range of a cycle, we set it to $-t$ where t is the index of that cycle and if it occurs again, it means that it belongs to that cycle and our work is done. Finally , as $R \subset S$, if S is itself a set of cycle, then we will remove the smallest cycle or self-loop in the S and it will become R. Here, is the implementation in C++,

```
1 // this function returns the set value with min count
2 int findminset (int arr [] , int N )
3 {
```

```
    map <int ,int > m ;
    int count = 0 , min_count = INT_MAX , set_val = -1;
    for ( int i =0; i < N ; i ++)
        if( arr [ i ] <= 0) { arr [ i ] = 0;}
        else { m [ arr [ i ]]++ ; count ++;}
    for ( auto x : m ) if( x . second < min_count ) { set_val = x .
     ↪  first ; min_count = x . second ;}
    if( count < N ) return -1;
    else return set_val ;
}
void findmaxset (int arr [] , int N , map <int ,int > f )
{
    queue <int > q ;
    vector <int > count ;
    int t = 2;
    bool flag ;
    // this part finds all cycles in the complete set S
    for ( int i =0; i < N ; i ++)
    {
        if( arr [ i ] == -1)
            {
            flag = true ;
            vector <int > pos ;
            q . push ( i ) ;
            while (! q . empty () )
            {
                int x = q . front () ; q. pop () ;
                if( arr [ x ] == -1) { arr [ x ] = -t ;}
                else if( arr [ x ] == -t ) { arr [ x ] = t ; pos .
                 ↪  push_back ( x ) ;}
                else { flag = false ;}
                if( arr [ f [ x +1] -1] == -1) { q . push ( f [ x
                 ↪  +1] -1) ; arr [ f [ x +1] - 1] = -t ;}
                else if( arr [ f [ x +1] - 1] == -t ) {( arr [ f [ x
                 ↪  +1] - 1] = t ) ; pos . push_back ( f [ x +1] -1)
                 ↪  ;}
                else { flag = false ;}
            }
        if( flag ) for( auto x : pos ) arr [ x ] = t ;
        else for( auto x : pos ) arr [ x ] = -t ;
        t ++;
        }
    }
```

```
42      // this part finds the min set and removes it from S
43      int mincount = findminset ( arr , N ) ;
44      for ( int i =0; i < N ; i ++)
45      {
46          if (( arr [ i ] != mincount ) && ( arr [ i ] != 0) ) { arr [
            ↪  i ] = 1;}
47          else { arr [ i ]=0;}
48      }
49  }
```

Time Complexity Analysis

1) findminset() : It contains two loop, one of which accesses every element once and second keeps track of no. of cycles. As, every element will be accesses once only and no. of cycles will be definitely less than equal to n, so Complexity will be $O(n)$
2) findmaxset() : Every element is accessed at most two times, one is while checking if it is a cycle, second is if it is a cycle then setting its index to the cycle index. So, complexity of the part without findminset() = $O(n)$ and complexity of findminset() = $O(n)$. So total complexity is $O(n)$.

14 Minimum Spanning Tree

Problem 14A. You are given a graph $G = (V, E)$ with positive edge weights, and a minimum spanning tree $T = (V, E')$ with respect to these weights ; you may assume G and T are given as adjacency lists. Now suppose the weight of a particular edge $e \in E$ is modified from $w(e)$ to a new value $w'(e)$. You wish to quickly update the minimum spanning tree T to reflect this change, without recomputing the entire tree from scratch. There are four cases. In each case give a linear-time algorithm for updating the tree.

(a) $e \notin E'$ and $w'(e) > w(e)$

(b) $e \notin E'$ and $w'(e) < w(e)$

(c) $e \in E'$ and $w'(e) < w(e)$

(d) $e \in E'$ and $w'(e) > w(e)$

Solution 14.1 — We are given a graph G with positive edge weights, and a minimum spanning tree T with edge set E'. A particular edge e's weight is modified to $w'(e)$. We have to modify the MST in the following cases :

(a) $e \notin E'$ and $w'(e) > w(e)$: No change is required in the minimum spanning tree. $e = (u, v)$ does not belong to the MST T, which implies $\exists$ a path P in the MST T whose length is $\leq w(e)$. Upon increasing its weight to $w'(e)$, the inequality is still satisfied. Hence, there is no need to modify the MST.

(b) $e \notin E'$ and $w'(e) < w(e)$: Consider the path $u \to p \ldots q \to v$ in the MST T that connects the two vertices u and v - the end points of the edge e. Upon adding the edge e to the MST T, a cycle is formed. This cycle can be found by simply using Breadth First Search to find the path from u to v int the MST T and just including the new edge e, ignoring weights. This takes time linear in the size of MST, which is $O(n)$ because there are $(n - 1)$ edges in a tree. Now, we remove the additional edge to maintain our tree structure. We simply iterate through the cycle and remove the one with the maximum weight. This step is again linear as it processes each edge exactly once. This greedy approach works because :

1) We have shown that the tree structure is maintained
2) The tree's weight is minimised

As all steps of the algorithm are accomplished in linear time, the overall time complexity amounts to $O(|V|)$.

(c) $e \in E'$ and $w'(e) < w(e)$: No change is required in the minimum spanning tree.

$e = (u, v)$ belongs to the MST T, which implies that $\nexists$ any edge / path that has a lower length than e and connects the two vertices u and v. Upon decreasing its weight further to $w'(e)$, the inequality is still satisfied. Hence, there is no need to modify the MST.

(d) $e \in E'$ and $w'(e) > w(e)$: It is possible that after update, the edge e might not be a part of the MST. Hence, we remove the edge e from the minimum spanning tree. Upon removal of this edge, we are left with two connected components. Iterating through the edges of the remaining almost-tree, we find the set of vertices T_u and T_v that represent the two connected components obtained after removing the edge (u, v). Now, we iterate through all the edges in G in ascending order of weight and choose the one that has endpoints in each of the two sets. As this step runs in $O(E)$ time, the overall time complexity reaches $O(V + E)$, which is still linear in the size of the graph G.

Problem 14B. The following statements may or may not be correct. In each case, either prove it (if it is correct) or give a counter example (if it isn't correct). Always assume that the graph $G = (V, E)$ is undirected. Do not assume that edge weights are distinct unless this is specifically stated.

(a) If graph G has more than $|V| - 1$ edges, and there is a unique heaviest edge, then this edge cannot be part of a MST

(b) If G has a cycle with a unique heaviest edge e, then e cannot be part of any MST

(c) Let e be any edge of minimum weight in G. Then e must be part of some MST

(d) If the lightest edge in a graph is unique, then it must be part of every MST

(e) If e is part of some MST of G, then it must be a lightest edge across some cut of G

(f) If G has a cycle with a unique lightest edge e, then e must be part of every MST

(g) The shortest-path tree computed by Dijkstra's algorithm is necessarily an MST

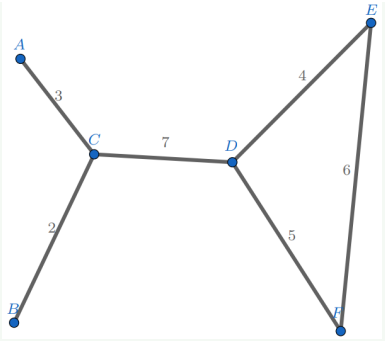
(h) The shortest path between two nodes is necessarily part of some MST

(i) Prim's algorithm works correctly when there are negative edges

(j) (For any $r > 0$, define an r-path to be a path whose edges all have weight $< r$.) If G contains an r-path from node s to t, then every MST of G must also contain an r-path from node s to node t

Solution 14.2 — Lets see each part one by one,

(a) This claim is false as the below counter example shows. The below graph has more than $|V| - 1$ edges. The edge CD is the maximum weight edge but it is part of every MST as it is a bridge for the above graph.



(b) This statement is true. We will prove this by an exchange argument. Let e be an edge which is the unique largest edge in a cycle of a graph G. Now let e be part of the MST and let the total cost of the MST which is the sum of the weights of the edges in the MST be C. Now there must be an edge adjacent to e in a cycle containing e in G that is not part of the MST. This is because an MST cannot contain a cycle and if such an edge does not exist then e itself is not part of any cycle in G which is not true. Let the edge be f. Surely f has lesser weight then e as per the definition of e. Now add the edge f to the MST. We get a uni-cyclic graph and the cycle in the graph contains both e and f. Now as e is part of a cycle it cannot be a bridge and therefore removing it does not disconnect the graph and we get a new spanning tree with cost $C + w_f - w_e$ Where w_f is the weight of the edge f which is lesser then the weight of edge e, w_e. Therefore the cost of the tree is $C - w_e + w_f < C$. Hence we have a spanning tree with lesser weight then the MST which is a contradiction.

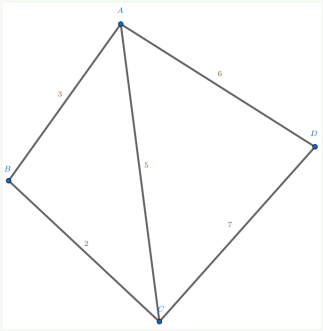
(c) This statement is true. All we need to show is that there exist a MST containing e, where e is the edge with minimum weight (not necessarily) unique. We will prove this with a simple exchange argument. If e is a bridge (cut edge) then it definitely must be part of any spanning tree and hence also the minimum spanning tree. Else let e be part of some cycle in G. Consider an MST which does not contain e and let its cost be C. It would contain some edge which shares an endpoint with e and also lies in a common cycle with e. There is surely one such edge as we can remove all the edges incident to a vertex v in G that are pairwise part of a cycle to disconnect the graph. Now let the edge be f. Now on adding e to the MST we get a uni-cyclic graph with e and f part of the cycle. As f is part of a cycle therefore it is not a bridge and we can remove it to get a spanning tree with cost $C' = C - w_f + w_e$, now $w_f \geq w_e$ as per the definition and hence the new cost is not greater than that of the MST i.e. $C' \leq C$ but as C is the minimum possible cost therefore $C' = C$ and we have an MST containing e.

(d) This statement is true. We will again prove this by an exchange argument leading to a contradiction. Let e be the unique minimum weight edge of the Graph G. If e is a bridge then it must be part of the MST. Otherwise e is part of some cycle in G. Let the MST of G not contain e and let its cost be C. Now by our argument in the

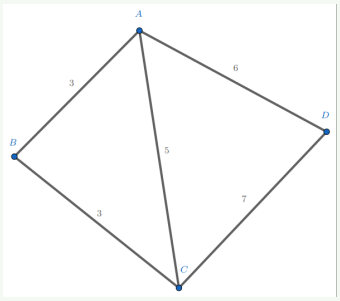
previous part some edge sharing a vertex with e and lying in a common cycle must lie in the minimum spanning tree. Let the edge be f. Now we can add e to the MST to construct a uni-cyclic graph whose cycle contains both f and e. Now as f lies in a cycle it cannot be a bridge and removing it creates a spanning tree with weight $C' = C - w_f + w_e$. Now by definition $w_e < w_f$, hence $C' < C$, and hence the new tree has a smaller cost. This contradicts the minimality of the MST. Hence proved.

(e) This statement is true. Consider the MST T corresponding to G, let its cost be C. Consider an edge $e \in E(T)$. Now e is a bridge in T and $T - \{e\}$ generates 2 forests say F_1 and F_2. Consider the subgraph G_1 induced by the vertices in $V(F_1)$ and G_2 induced by the vertices in $V(F_2)$. Consider all the edges between the the vertices in G_1 and G_2. Let the corresponding set be S. Then S is a cut in G and $G - S = G_1 \cup G_2$ and $G_1 \cap G_2 = \phi$. Now we claim that $e \in S$ is the lightest edge in S. The proof is easy. If e is not the lightest edge then there exist some edge $f \in S$ such that $W_f < w_e$ and removing e and adding f to the forests F_1 and F_2 connects to give a tree with cost $C' = C - w_e + w_f < C$. This contradicts the minimality of the MST. Hence e must be the lightest edge in the cut S. Hence Proved.

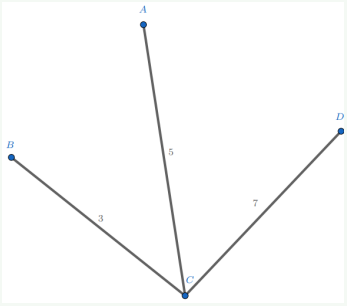
(f) Consider the below graph. The edge AC has weight 5 and is the unique lightest edge in the cycle $ADCA$. But the MST for the graph is the path $C - B - A - D$ which has cost 11 and does not contain the edge AC. Hence this graph is a counter example to the above claim.



(g) Consider the below graph.

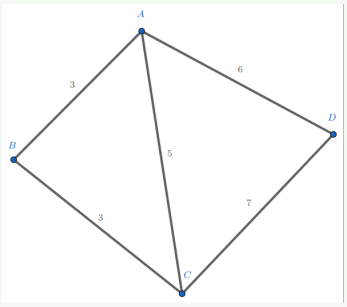


Running Dijkstra'a single source shortest path algorithm from the vertex C we get the following tree -



The weight of the above tree is 15 whereas the weight of the minimum spanning tree is 12. Hence the above claim is false.

(h) Consider the above graph. The shortest path from vertex A to C is the edge AC with weight 5. Now the MST of the graph G does not contain the edge AC. Since the MST of the graph is unique in this case therefore the above claim is false as the shortest path between A and C does not lie in any MST.



(i) This statement is true. Prim's algorithm works if the edges are negative. This is easy to see. In Prim's Algorithm we maintain 2 set of vertices set A and set B. Set A initially has a single vertex and set B has rest of the vertices. At each iteration the edge with minimum weight between the 2 sets is added to the MST and the corresponding end point is added to set A. The algorithm continues until the set B becomes empty. Now note that at any step we only care about finding the minimum weight edge between the set A and set B, it does not matter what the exact weight is. Thus if we add a constant c to the weight of every edge we get a isomorphic tree as the MST whose weight is $(n-1)c$ more than the MST for the original graph. Therefore we can make all the edges as positive by adding a suitable constant to the weight for every edge and we get the MST for the this new graph by Prims's Algorithm. We can now retrieve the MST for the original graph by just subtracting c from the weight of every edge in the new MST. Hence Prim's Algorithm works for graphs with negative edges.

(j) This statement is true. We will prove by contradiction. Consider a graph $G = (V, E)$. Let $s, t \in V$ be two vertices. Now consider any r path P_1 between the 2 vertices. Therefore the weight of every edge on the path P_1 is $< r$. Now let some MST of G with cost C not contain an r path between vertices s and t. Consider the path P_2 between s and t in the MST (there is one unique path). Now since P_1 is an r path between s and t, none of the edges in P_2 lie in P_1. Now there are some edges in P_1 not in the MST because if all are present then we have an r path in the MST which is P_1 itself. Consider $P_1 \cup P_2$ in G. Since they cannot be the same they enclose 1 or many cycles. We claim that one of the cycles contains an edge with weight $\geq r$. This is simple to see as P_2 does contain one such edge and that edge cannot lie on the common path of P_1 and P_2, hence it must lie on some cycle enclosed by P_1 and P_2. There must be atleast one edge of P_1 in the given cycle. Let the edge be e. Remove all added edges except e from the graph and what we are left with is a uni-cyclic graph whose cycle contains both e and the edge in P_2 with weight $\geq r$ say f. Now since f lies in a cycle it cannot be a bridge and removing it cannot disconnect the uni-cyclic graph. Removing it we get a spanning tree with cost $C - w_f + w_e$. Now $w_e < r \leq w_f$. Therefore the cost of the spanning tree is $C' = C - w_f + w_e < C$ which contradicts the minimality of the MST.

15 Dynamic Programming

Problem 15A. Satya is in charge of establishing a new testing center for the Standardized Awesomeness Test (SAT), and found an old conference hall that is perfect. The conference hall has n rooms of various sizes along a single long hallway, numbered in order from 1 through n. Satya knows exactly how many students fit into each room, and he wants to use a subset of the rooms to host as many students as possible for testing. Unfortunately, there have been several incidents of students cheating at other testing centers by tapping secret codes through walls. To prevent this type of cheating, Satya can use two adjacent rooms only if he demolishes the wall between them. The city's chief architect has determined that demolishing the walls on both sides of the same room would threaten the building's structural integrity. For this reason, Satya can never host students in three consecutive rooms. Describe an efficient algorithm that computes the largest number of students that Satya can host for testing without using three consecutive rooms. The input to your algorithm is an array $S[1 \ldots n]$, where each $S[i]$ is the (non-negative integer) number of students that can fit in room i.

Solution 15.1 — For any index i and any integer $c \in \{0, 1, 2\}$, let $MaxStudents(i, c)$ denote the maximum number of students that can be hosted for testing in rooms i through n, assuming exactly c rooms immediately before room i are already being used. We need to compute $MaxStudents(1, 0)$. This function can be described by the following recurrence :

$$MaxStudents(i, c) = \begin{cases} 0 & \text{if } i > n \\ MaxStudents(i+1, 0) & \text{if } c = 2 \\ \max \begin{pmatrix} S[i] + MaxStudents(i+1, c+1) \\ MaxStudents(i+1, 0) \end{pmatrix} & \text{otherwise} \end{cases}$$

We can memoize this function into an array $MaxStudents[1 \ldots n, 0 \ldots 2]$, which we can fill by decreasing i in the outer loop and considering c in any order in the inner loop. The resulting algorithm runs in $O(n)$ time.

An alternate solution : for any index i, let $MaxStudents(i)$ denote the maximum number of students that can be hosted for testing in rooms i through n, assuming room $i - 1$ is not occupied. We need to compute $MaxStudents(1)$. This function can be described by the

following recurrence :

$$MaxStudents(i) = \begin{cases} 0 & \text{if } i > n \\ S[n] & \text{if } i = n \\ S[n-1] + S[n] & \text{if } i = n-1 \\ \max \begin{pmatrix} MaxStudents(i+1) \\ S[i] + MaxStudents(i+2) \\ S[i] + S[i+1] + MaxStudents(i+3) \end{pmatrix} & \text{otherwise} \end{cases}$$

We can memoize this function into an array $MaxStudents[1 \ldots n]$, which we can fill by decreasing i in $O(n)$ time.

Problem 15B. As a typical overworked college student, you occasionally pull all-nighters to get more work done. Painful experience has taught you that the longer you stay awake, the less productive you are. Suppose there are n days left in the semester. For each of the next n days, you can either stay awake and work, or you can sleep. You have an array $Score[1 \ldots n]$, where $Score[i]$ is the (always positive) number of points you will earn on day i if you are awake and well-rested. However, staying awake for several days in a row has a price : Each consecutive day you stay awake cuts the quality of your work in half. Thus, if you are awake on day i, and you most recently slept on day $i-k$, then you will actually earn $Score[i]/2^{k-1}$ points on day i. (You've already decided to sleep on day 0.) For example, suppose $n = 6$ and $Score = [3, 7, 4, 3, 9, 1]$.

- If you work on all six days, you will earn 3 + 7/2 + 4/4 + 3/8 + 9/16 + 1/32 = 8.46875 points
- If you work only on days 1, 3, and 5, you will earn 3 + 4 + 9 = 16 points
- If you work only on days 2, 3, 5, and 6, you will earn 7 + 4/2 + 9 + 1/2 = 18.5 points

Design and analyze an algorithm that computes the maximum number of points you can earn, given the array$Score[1 \ldots n]$ as input. For example, given the input array [3, 7, 4, 3, 9, 1], your algorithm should return the number 18.5.

Solution 15.2 — For any integers i and k such that $i \geq k$, let $MaxScore(i, k)$ be the maximum total score I can earn on assignments i through n,assuming I slept on day $i-k-1$ and stayed awake for the k consecutive days $i-k$ through $i-1$. We need to compute $MaxScore(1, 0)$. This function obeys the following recurrence : Either I stay awake on day i (making that my $(k+1)$-th consecutive day awake) and earn the penalized score for that day, or I sleep on day i and earn no points. We can memoize

$$MaxScore(i, k) = \begin{cases} 0 & \text{if } i > n \\ \max \left(\begin{cases} MaxScore(i+1, k+1) + Score[i]/2^k \\ MaxScore(i+1, 0) \end{cases} \right) & \text{otherwise} \end{cases}$$

this function into a two-dimensional array $MaxScore[1 \ldots n, 0 \ldots n]$ indexed by i and then

k. We can fill this array by decreasing i in the outer loop, and by increasing k in the inner loop, in $O(n^2)$ time.

An alternate solution : For any integer i, let$MaxScore(i)$ be the maximum score total score I can earn on assignments i through n, assuming I sleep on day $i-1$. We need to compute $MaxScore(1)$. This function obeys the following recurrence.

$$MaxScore(i) = \begin{cases} 0 & \text{if } i > n \\ \max_{l=0}^{n-i} \left(\sum_{j=0}^{l-1} \frac{Score[i+j]}{2^j} + MaxScore(i+l+1) \right) & \text{otherwise} \end{cases}$$

The expression in braces is the maximum score I can earn if I work for l consecutive days starting on day i, and then sleep on day $i+l$. The recurrence tries all possible values of l. We can memoize this function into a one-dimensional array $MaxScore[1 \ldots n]$. We can fill this array from right to let (decreasing i). A straightforward implementation runs in $O(n^3)$ time : For each index i, we compute $MaxScore[i]$ using two nested loops (over l and j). We can make this algorithm faster by defining a helper function

$$Sum(i,l) := \sum_{j=0}^{l-1} \frac{Score[i+j]}{2^j}$$

This helper function gives us a simpler recurrence for *MaxScore*:

$$MaxScore(i) = \max_{l=0}^{n-i} \{Sum(i,l) + MaxScore(i+l+1)\}$$

The helper function itself satisfies the recurrence

$$Sum(i,l) = \begin{cases} 0 & \text{if } l = 0 \\ Sum(i,l-1) + \frac{Score[i+l-1]}{2^l} & \text{otherwise} \end{cases}$$

We can memoize this function into a two-dimensional array $Sum[1 \ldots m, 0 \ldots n]$, which we can fill by considering the rows (indexed by i) in any order in the outer loop, and filling each each row from right to left (decreasing l). Thus we can compute $Sum[i,l]$ for all i and l in $O(n^2)$ time. After this preprocessing step, we can replace the innermost loop in our algorithm for $MaxScore$ with an array lookup. The optimized algorithm runs in $O(n^2)$ time.

Problem 15C. This problem has two subparts,

(a) Any string can be decomposed into a sequence of palindromes. For example, the string BUBBASEESABANANA ("Bubba sees a banana.") can be broken into palindromes in the following ways (and 65 others):

BUB • BASEESAB • ANANA

B • U • BB • ASEESA • B • ANANA

BUB • B • A • SEES • ABA • N • ANA

B • U • BB • A • S • EE • S • A • B • A • NAN • A

B • U • B • B • A • S • E • E • S • A • B • A • N • A • N • A

Describe and analyze an efficient algorithm to find the smallest number of palindromes that make up a given input string. For example, given the input string BUBBASEESABANANA, your algorithm should return 3.

(b) A ***metapalindrome*** is a decomposition of a string into a sequence of palindromes, such that the sequence of palindrome lengths is itself a palindrome. For example, the string BOBSMAMASEESAUKULELE ("Bob's mama sees a ukulele") has the following metapalindromes (among others):

BOB • S • MAM • ASEESA • UKU • L • ELE

B • O • B • S • M • A • M • A • S • E • E • S • A • U • K • U • L • E • L • E

The length sequences of these metapalindromes are $(3, 1, 3, 6, 3, 1, 3)$ and $(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)$; notice that both of these sequences are themselves palindromes.

Describe and analyze an efficient algorithm to find the smallest number of palindromes in any metapalindrome for a given string. For example, given the input string BOBSMAMASEESAUKULELE, your algorithm should return 7.

Solution 15.3 — Let $A[1 \ldots n]$ be the input string. We define two functions :

- *IsPal?*(i, j) is TRUE if the substring $A[i..j]$ is a palindrome, and FALSE otherwise.
- *MinPals*(k) is the minimum number of palindromes that make up the suffix $A[k..n]$.

We need to compute *MinPals*(1).

Every string with length at most 1 is a palindrome. A string of length 2 or more is a palindrome if and only if its first and last characters are equal *and* the rest of the string is a palindrome. Thus:

$$IsPal?(i,j) = \begin{cases} \text{TRUE} & \text{if } i \geq j \\ (A[i] = A[j]) \wedge IsPal?(i+1, j-1) & \text{otherwise} \end{cases}$$

We can memoize this function into a two-dimensional array $IsPal?[1..n, 0..n]$. Each entry $IsPal[i,j]$ depends only on $IsPal[i+1, j-1]$. Thus, we can fill this array row-by-row, from the bottom row upward, in $O(n^2)$ time.

The empty string can be partitioned into zero palindromes. Otherwise, the best palindrome decomposition has at least one palindrome. If the first palindrome in the *optimal* decomposition of $A[1..n]$ ends at index ℓ, the remainder must be the *optimal* decomposition for the remaining characters $A[\ell+1..n]$. The following recurrence considers all possible values of ℓ.

$$MinPals(k) = \begin{cases} 0 & \text{if } k > n \\ 1 + \min\left\{ MinPals(\ell+1) \;\middle|\; k \leq \ell \leq n \text{ and } IsPal?(k,\ell) \right\} & \text{otherwise} \end{cases}$$

We can memoize this function into a one-dimensional array $MinPals[1..n]$. Each entry $MinPals[k]$ depends only on entries $MinPals[\ell+1]$ with $\ell \geq k$. Thus, we can fill this array from right to left, in $O(n^2)$ time.

The subroutine COMPUTEISPAL? runs in $O(n^2)$ time, and the rest of the algorithm also clearly runs in $O(n^2)$ time. So the total running time is $\boldsymbol{O(n^2)}$.

If we had used the obvious iterative algorithm to test whether each substring is a palindrome, instead of precomputing the array *IsPal?*, our algorithm would have run in $O(n^3)$ time. ■

Solution (pseudocode): The following algorithms both run in $\boldsymbol{O(n^2)}$ ***time***. This first algorithm precomputes a boolean array *IsPal?*, where $IsPal?[i,j] = \text{TRUE}$ if and only if the substring $A[i..j]$ is a palindrome.

```
COMPUTEISPAL?(A[1..n]):
  for i ← n down to 1
      IsPal?[i, i−1] ← TRUE
      IsPal?[i, i] ← TRUE
      for j ← i+1 to n
          if A[i] = A[j]
              IsPal?[i, j] ← IsPal?[i+1, j−1]
          else
              IsPal?[i, j] ← FALSE
```

Here is the main algorithm; $MinPals[k]$ stores the minimum number of palindromes whose concatenation is the suffix $A[k..n]$.

```
MINPALS(A[1..n]):
  COMPUTEISPAL?(A[1..n])
  MinPals[n+1] ← 0
  for k ← n down to 1
      MinPals[k] ← ∞
      for ℓ ← k to n
          if ISPAL?[k, ℓ]
              MinPals[k] ← min{MinPals[k], 1 + MinPals[ℓ+1]}
  return MinPals[1]
```

■

V

Additional

16 Errata

- Section 1.2 Note that n is odd, however earlier it was written as even.
- Section 1.2 Note that the inequality sign is reversed and the correct relation is $|S| > |S^{'}|$ while earlier it was written as $|S| < |S^{'}|$.
- Section 5.4.ii.2 Note that only the tree rooted at r changes and not its size while earlier it was written that size of tree rooted at r changes in ***Inductive hypothesis***
- Section 11.2 Note that only in a walk, the number of edges or vertices can be repeated any number of times, while not in a path while earlier the word walk was wrongly written as path
- Section 12.2 Note that in Step 2, for the recursive definition of $c[i, j]$ when $i, j > 0$ and $x_i \neq y_j$ is $c[i, j] = \max\{c[i, j-1], c[i-1, j]\}$ while earlier it was wrongly written as $c[i, j] = \max\{c[i, j-1] + c[i-1, j]\}$
- Section 9.4 Note that in line 5 of algorithm the condition is FIND-SET(u) $\neq$ FIND-SET(v) while earlier it was wrongly written as FIND-SET(u) = FIND-SET(v)