Solutions for Introduction to algorithms second edition

Philip Bille

The author of this document takes absolutely no responsibility for the contents. This is merely a vague suggestion to a solution to some of the exercises posed in the book Introduction to algorithms by Cormen, Leiserson and Rivest. It is very likely that there are *many* errors and that the solutions are wrong. If you have found an error, have a better solution or wish to contribute in some constructive way please send a message to `beetle@it.dk`.

It is important that you try *hard* to solve the exercises on your own. Use this document only as a last resort or to check if your instructor got it all wrong.

Please note that the document is under construction and is updated only sporadically. Have fun with your algorithms.

Best regards,
Philip Bille

Last update: December 9, 2002

1.2 − 2

Insertion sort beats merge sort when $8n^2 < 64n \lg n$, $\Rightarrow n < 8 \lg n$, $\Rightarrow 2^{n/8} < n$. This is true for $2 \leqslant n \leqslant 43$ (found by using a calculator).

Rewrite merge sort to use insertion sort for input of size 43 or less in order to improve the running time.

1 − 1

We assume that all months are 30 days and all years are 365.

	1 second	1 minute	1 hour	1 day	1 month	1 year	1 century
$\lg n$	2^{10^6}	$2^{6 \cdot 10^7}$	$2^{36 \cdot 10^8}$	$2^{864 \cdot 10^8}$	$2^{2592 \cdot 10^9}$	$2^{94608 \cdot 10^{10}}$	$2^{94608 \cdot 10^{12}}$
$\sqrt{n}$	10^{12}	$36 \cdot 10^{14}$	$1296 \cdot 10^{16}$	$746496 \cdot 10^{16}$	$6718464 \cdot 10^{18}$	$8950673664 \cdot 10^{20}$	$8950673664 \cdot 10^{24}$
n	10^6	$6 \cdot 10^7$	$36 \cdot 10^8$	$864 \cdot 10^8$	$2592 \cdot 10^9$	$94608 \cdot 10^{10}$	$94608 \cdot 10^{12}$
$n \lg n$	62746	2801417	??	??	??	??	??
n^2	10^3	24494897	$6 \cdot 10^4$	293938	1609968	30758413	307584134
n^3	10^2	391	1532	4420	13736	98169	455661
2^n	19	25	31	36	41	49	56
$n!$	9	11	12	13	15	17	18

$2.1-2$

In line 5 of INSERTION-SORT alter $A[i] > key$ to $A[i] < key$ in order to sort the elements in nonincreasing order.

$2.1-3$

Algorithm 1 LINEAR-SEARCH(A, v)

```
Input: $A = \langle a_1, a_2, \dots a_n \rangle$ and a value $v$.
Output: An index $i$ such that $v = A[i]$ or nil if $v \notin A$
for $i \leftarrow 1$ to $n$ do
  if $A[i] = v$ then
    return $i$
  end if
end for
return nil
```

As a loop invariant we say that none of the elements at index $A[1, \dots, i-1]$ are equal to v. Clearly, all properties are fullfilled by this loop invariant.

$2.2-1$

$n^3/1000 - 100n^2 - 100n + 3 = \Theta(n^3)$.

$2.2-2$

Assume that FIND-MIN(A, r, s) returns the index of the smallest element in A between indices r and s. Clearly, this can be implemented in $O(s-r)$ time if $r \geqslant s$.

Algorithm 2 SELECTION-SORT(A)

```
Input: $A = \langle a_1, a_2, \dots a_n \rangle$
Output: sorted $A$.
for $i \leftarrow 1$ to $n-1$ do
  $j \leftarrow$ FIND-MIN$(A, i, n)$
  $A[j] \leftrightarrow A[i]$
end for
```

As a loop invariant we choose that $A[1, \dots, i-1]$ are sorted and all other elements are greater than these. We only need to iterate to $n-1$ since according to the invariant the nth element will then the largest.

The n calls of FIND-MIN gives the following bound on the time complexity:

$$\Theta\left(\sum_{i=1}^{n} i\right) = \Theta(n^2)$$

This holds for both the best- and worst-case running time.

$2.2-3$

Given that each element is equally likely to be the one searched for and the element searched for is present in the array, a linear search will on the average have to search through half the elements. This is because half the time the wanted element will be in the first half and half the time it will be in the second half. Both the worst-case and average-case of LINEAR-SEARCH is $\Theta(n)$.

2.2 − 4

One can modify an algorithm to have a best-case running time by specializing it to handle a best-case input efficiently.

2.3 − 5

A recursive version of binary search on an array. Clearly, the worst-case running time is $\Theta(\lg n)$.

Algorithm 3 BINARY-SEARCH(A, v, p, r)

```
Input: A sorted array $A$ and a value $v$.
Output: An index $i$ such that $v = A[i]$ or nil.
if $p \geqslant r$ and $v \neq A[p]$ then
   return nil
end if
$j \leftarrow A[\lfloor (r-p)/2 \rfloor]$
if $v = A[j]$ then
   return $j$
else
   if $v < A[j]$ then
      return BINARY-SEARCH$(A, v, p, j)$
   else
      return BINARY-SEARCH$(A, v, j, r)$
   end if
end if
```

2.3 − 7

Give a $\Theta(n \lg n)$ time algorithm for determining if there exist two elements in an set S whose sum is exactly some value x.

Algorithm 4 CHECKSUMS(A, x)

```
Input: An array $A$ and a value $x$.
Output: A boolean value indicating if there is two elements in $A$ whose sum is $x$.
$A \leftarrow$ SORT$(A)$
$n \leftarrow length[A]$
for $i \leftarrow$ to $n$ do
   if $A[i] \geqslant 0$ and BINARY-SEARCH$(A, A[i] - x, 1, n)$ then
      return true
   end if
end for
return false
```

Clearly, this algorithm does the job. (It is assumed that **nil** cannot be true in the **if**-statement.)

3.1 − 1

Let $f(n)$, $g(n)$ be asymptotically nonnegative. Show that $\max(f(n), g(n)) = \Theta(f(n) + g(n))$. This means that there exists positive constants c_1, c_2 and n_0 such that,

$$0 \leqslant c_1(f(n) + g(n)) \leqslant \max(f(n), g(n)) \leqslant c_2(f(n) + g(n))$$

for all $n \geqslant n_0$.

Selecting $c_2 = 1$ clearly shows the third inequality since the maximum must be smaller than the sum. c_1 should be selected as $1/2$ since the maximum is always greater than the weighted average of $f(n)$ and $g(n)$. Note the significance of the "asymptotically nonnegative" assumption. The first inequality could not be satisfied otherwise.

3.1 − 4

$2^{n+1} = O(2^n)$ since $2^{n+1} = 2 \cdot 2^n \leqslant 2 \cdot 2^n$! However 2^{2n} is not $O(2^n)$: by definition we have $2^{2n} = (2^n)^2$ which for no constant c asymptotically may be less than or equal to $c \cdot 2^n$.

3 − 4

Let $f(n)$ and $g(n)$ be asymptotically positive functions.

a. No, $f(n) = O(g(n))$ does not imply $g(n) = O(f(n))$. Clearly, $n = O(n^2)$ but $n^2 \neq O(n)$.

b. No, $f(n) + g(n)$ is not $\Theta(\min(f(n), g(n)))$. As an example notice that $n + 1 \neq \Theta(\min(n, 1)) = \Theta(1)$.

c. Yes, if $f(n) = O(g(n))$ then $\lg(f(n)) = O(\lg(g(n)))$ provided that $f(n) \geqslant 1$ and $\lg(g(n)) \geqslant 1$ are greater than or equal 1. We have that:

$$f(n) \leqslant cg(n) \Rightarrow \lg f(n) \leqslant \lg(cg(n)) = \lg c + \lg g(n)$$

To show that this is smaller than $b \lg g(n)$ for some constant b we set $\lg c + \lg g(n) = b \lg g(n)$. Dividing by $\lg g(n)$ yields:

$$b = \frac{\lg(c) + \lg g(n)}{\lg g(n)} = \frac{\lg c}{\lg g(n)} + 1 \leqslant \lg c + 1$$

The last inequality holds since $\lg g(n) \geqslant 1$.

d. No, $f(n) = O(g(n))$ does not imply $2^{f(n)} = O(2^{g(n)})$. If $f(n) = 2n$ and $g(n) = n$ we have that $2n \leqslant 2 \cdot n$ but not $2^{2n} \leqslant c2^n$ for any constant c by exercise 3.1 − 4.

e. Yes and no, if $f(n) < 1$ for large n then $f(n)^2 < f(n)$ and the upper bound will not hold. Otherwise $f(n) > 1$ and the statement is trivially true.

f. Yes, $f(n) = O(g(n))$ implies $g(n) = \Omega(f(n))$. We have $f(n) \leqslant cg(n)$ for positive c and thus $1/c \cdot f(n) \leqslant g(n)$.

g. No, clearly $2^n \nleqslant c2^{n/2} = c\sqrt{2^n}$ for any constant c if n is sufficiently large.

h. By a small modification to exercise 3.1−1 we have that $f(n)+o(f(n)) = \Theta(\max(f(n), o(f(n)))) = \Theta(f(n))$.

$4.1 - 1$

Show that $T(n) = T(\lceil n/2 \rceil) + 1$ is $O(\lg n)$. Using substitution we want to prove that $T(n) \leqslant c\lg(n - b)$. Assume this holds for $\lceil n/2 \rceil$. We have:

$$\begin{aligned}
T(n) &\leqslant c\lg(\lceil n/2 - b \rceil) + 1 \\
&< c\lg(n/2 - b + 1) + 1 \\
&= c\lg(\frac{n - 2b + 2}{2}) + 1 \\
&= c\lg(n - 2b + 2) - c\lg 2 + 1 \\
&\leqslant c\lg(n - b)
\end{aligned}$$

The last inequality requires that $b \geqslant 2$ and $c \geqslant 1$.

$4.2 - 1$

Determine an upper bound on $T(n) = 3T(\lfloor n/2 \rfloor) + n$ using a recursion tree. We have that each node of depth i is bounded by $n/2^i$ and therefore the contribution of each level is at most $(3/2)^i n$. The last level of depth $\lg n$ contributes $\Theta(3^{\lg n}) = \Theta(n^{\lg 3})$. Summing up we obtain:

$$\begin{aligned}
T(n) &= 3T(\lfloor n/2 \rfloor) + n \\
&\leqslant n + (3/2)n + (3/2)^2 n + \cdots + (3/2)^{\lg n - 1} n + \Theta(n^{\lg 3}) \\
&= n \sum_{i=0}^{\lg n - 1} (3/2)^i + \Theta(n^{\lg 3}) \\
&= n \cdot \frac{(3/2)^{\lg n} - 1}{(3/2) - 1} + \Theta(n^{\lg 3}) \\
&= 2(n(3/2)^{\lg n} - n) + \Theta(n^{\lg 3}) \\
&= 2n\frac{3^{\lg n}}{2^{\lg n}} - 2n + \Theta(n^{\lg 3}) \\
&= 2 \cdot 3^{\lg n} - 2n + \Theta(n^{\lg 3}) \\
&= 2n^{\lg 3} - 2n + \Theta(n^{\lg 3}) \\
&= \Theta(n^{\lg 3})
\end{aligned}$$

We can prove this by substitution by assumming that $T(\lfloor n/2 \rfloor) \leqslant c\lfloor n/2 \rfloor^{\lg 3} - c\lfloor n/2 \rfloor$. We obtain:

$$\begin{aligned}
T(n) &= 3T(\lfloor n/2 \rfloor) + n \\
&\leqslant 3c\lfloor n/2 \rfloor^{\lg 3} - c\lfloor n/2 \rfloor + n \\
&\leqslant \frac{3cn^{\lg 3}}{2^{\lg 3}} - \frac{cn}{2} + n \\
&\leqslant cn^{\lg 3} - \frac{cn}{2} + n \\
&\leqslant cn^{\lg 3}
\end{aligned}$$

Where the last inequality holds for $c \geqslant 2$.

4.2 − 3

Draw the recursion tree of $T(n) = 4T(\lfloor n/2 \rfloor) + cn$. The height of the tree is $\lg n$, the out degree of each node will be 4 and the contribution of the ith level will be $4^i \lfloor cn/2^i \rfloor$. The last level contributes $4^{\lg n}\Theta(1) = \Theta(n^2)$. Hence we have a bound on the sum given by:

$$\begin{aligned}
T(n) &= 4T(\lfloor n/2 \rfloor) + cn \\
&= \sum_{i=0}^{\lg n - 1} 4^i \cdot \lfloor cn/2^i \rfloor + \Theta(n^2) \\
&\leqslant \sum_{i=0}^{\lg n - 1} 4^i \cdot cn/2^i + \Theta(n^2) \\
&= cn \sum_{i=0}^{\lg n - 1} 2^i + \Theta(n^2) + \Theta(n^2) \\
&= cn \cdot \frac{2^{\lg n} - 1}{2 - 1} + \Theta(n^2) \\
&= \Theta(n^2)
\end{aligned}$$

Using the substitution method we can verify this bound. Assume the following clever induction hypothesis. Let $T(\lfloor n/2 \rfloor) \leqslant c\lfloor n/2 \rfloor^2 - c\lfloor n/2 \rfloor$. We have:

$$\begin{aligned}
T(n) &= 4T(\lfloor n/2 \rfloor) + cn \\
&\leqslant 4(c\lfloor n/2 \rfloor^2 - c\lfloor n/2 \rfloor) + cn \\
&< 4c(n/2)^2 - 4cn/2 + cn \\
&= cn^2 - 2cn + cn \\
&= cn^2 - cn
\end{aligned}$$

4.3 − 1

Use the master method to find bounds for the following recursions. Note that $a = 4, b = 4$ and $n^{\log_2 4} = n^2$

- $T(n) = 4T(n/2) + n$. Since $n = O(n^{2-\epsilon})$ case 1 applies and we get $T(n) = \Theta(n^2)$.
- $T(n) = 4T(n/2) + n^2$. Since $n^2 = \Theta(n^2)$ we have $T(n) = \Theta(n^2 \lg n)$.
- $T(n) = 4T(n/2) + n^3$. Since $n^3 = \Omega(n^{2+\epsilon})$ and $4(n/2)^3 = 1/2n^3 \leqslant cn^3$ for some $c < 1$ we have that $T(n) = \Theta(n^3)$.

6.1 − 1

There is a most $2^{h+1} - 1$ vertices in a complete binary tree of height h. Since the lower level need not be filled we may only have 2^h vertices.

6.1 − 2

Since the height of an n-element heap must satisfy that $2^h \leqslant n \leqslant 2^{h+1} - 1 < 2^{h+1}$. we have $h \leqslant \lg n < h + 1$. h is an integer so $h = \lfloor \lg n \rfloor$.

6.1 − 3

The max-heap property insures that the largest element in a subtree of a heap is at the root of the subtree.

6.1 − 4

The smallest element in a max-heap is always at a leaf of the tree assuming that all elements are distinct.

6.1 − 6

No, the sequence $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$ is not a max-heap.

6.2 − 6

Setting the root to 0 and all other nodes to 1, will cause the 0 to propagate to bottom of the tree using at least $\lg n$ operations each costing $O(1)$. Hence we have a $\Omega(\lg n)$ lower bound for MAX-HEAPIFY.

6.4 − 4

Show that the worst-case running time of heapsort is $\Omega(n \lg n)$. This is clear since sorting has a lower bound of $\Omega(n \lg n)$

6.5 − 3

To support operations for a min-heap simply swap all comparisons between keys or elements of the heap in the max-heap implementation.

6.5 − 4

Since the heap data structure is represented by an array and deletions are implemented by reducing the size of the array there may be undefined values in the array past the end of the heap. Therefore it is essential that the MAX-HEAP-INSERT sets the key of the inserted node to $-\infty$ such that HEAP-INCREASE-KEY does not fail.

6.5 − 5

By the following loop invariant we can prove the correctness of HEAP-INCREASE-KEY:

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

Initialization: Before the first iteration of the while the only change of the max-heap is that $A[i]$ is increased an may therefore violate the max-heap property.

Maintenance: Immediately before the iteration i and the child that violated the max-heap property has been exchanged thus restoring the max-heap property between these. This can only destroy the max-heap property between i and the parent of i.

Termination: The termination condition of the **while** states that at the end of the iteration the max-heap property between i and its parent is restored or the i is the root of the heap.

We see by the loop invariant that the heap property is restored at end of the iteration.

$6.5-7$

Algorithm 5 HEAP-DELETE(A, i)

```
Input: A max-heap $A$ and integers $i$.
Output: The heap $A$ with the element a position $i$ deleted.
$A[i] \leftrightarrow A[\mathit{heap\text{-}size}[A]]$
$\mathit{heap\text{-}size}[A] \leftarrow \mathit{heap\text{-}size}[A] - 1$
$\mathit{key} \leftarrow A[i]$
if $\mathit{key} \leqslant A[\text{PARENT}(i)]$ then
  MAX-HEAPIFY$(A, i)$
else
  while $i > 1$ and $A[\text{PARENT}(i)] < \mathit{key}$ do
    $A[i] \leftrightarrow A[\text{PARENT}(i)]$
    $i \leftarrow \text{PARENT}(i)$
  end while
end if
```

$6.5-8$

Given k sorted lists with a total of n elements show how to merge them in $O(n \lg k)$ time. Insert all k elements a position 1 from each list into a heap. Use EXTRACT-MAX to obtain the first element of the merged list. Insert element at position 2 from the list where the largest element originally came from into the heap. Continuing in this fashion yields the desired algorithm. Clearly the running time is $O(n \lg k)$.

$6-2$

a. A d-ary heap can be implemented using a dimensional array as follows. The root is kept in $A[1]$, its d children are kept in order in $A[2]$ through $A[d+1]$ and so on. The procedures to map a node with index i to its parent and its jth child are given by:

```
D-ARY-PARENT$(i)$
return $\lfloor (i-2)/d + 1 \rfloor$

D-ARY-CHILD$(i, j)$
return $d(i-1) + j + 1$
```

b. Since each node has d children the height of the tree is $\Theta(\log_d n)$.

c. The HEAP-EXTRACT-MAX algorithm given in the text works fine for d-ary heaps; the problem is MAX-HEAPIFY. Here we need to compare the argument node to all its children. This takes $\Theta(d \log_d n)$ and dominates the overall time spent by HEAP-EXTRACT-MAX.

d. The MAX-HEAP-INSERT given in the text works fine as well. The worst-case running time is the height of the heap, that is $\Theta(\log_d n)$.

e. The HEAP-INCREASE-KEY algorithm given in the text works fine.

$7.1 - 2$

When all the elements in A are the same, notice that the comparison in line 4 of PARTITION is always satified and i therefore is incremented in each iteration. Since initially $i \leftarrow p - 1$ and $i + 1$ is returned the returned value is $r - 1$.

To make PARTITION return $(p + r)/2$ when all elements are the same, simply modify the algorithm to check for this case explicitly.

$7.1 - 3$

The running time of PARTITION is $\Theta(n)$ since each iteration of the **for** loop involves a constant number of operations and there is $\Theta(n)$ iterations in total.

$7.1 - 4$

To make QUICKSORT sort in nonincreasing order replace the $\leqslant$ comparison in PARTITION line 4 with $\geqslant$.

$7.2 - 2$

If the elements in A are the same, then by exercise $7.1 - 2$ the returned element from each call to PARTITION(A, p, r) is $r - 1$ thus yielding the worst-case partitioning. The total running time is easily seen to be $\Theta(n^2)$.

$7.2 - 3$

If the elements in A are distinct and sorted in decreasing order then, as in the previous exercise, we have worst-case partitioning. The running time is again $\Theta(n^2)$.

$7 - 4$

a. Clearly, the QUICKSORT' does exactly the same as the original QUICKSORT and therefore works correctly.

b. Worst-case partitioning can cause the stack depth of QUICKSORT' to be $\Theta(n)$.

c. If we recursively call QUICKSORT' on the smallest subarray returned by PARTITION we will avoid the problem and retain a $O(\lg n)$ bound on the stack depth.

8.2 − 3

COUNTING-SORT will work correctly no matter what order A is processed in, however it is not stable. The modification to the **for** loop actually causes numbers with the same value to appear in reverse order in the output. This can seen by running a few examples.

8.2 − 4

Given n integers from 1 to k show how to count the number of elements from a to b in $O(1)$ time with $O(n+k)$ preprocessing time. As shown in COUNTING-SORT we can produce an array C such that $C[i]$ contains the number of elements less than or equal to i. Clearly, $C[b] - C[a]$ gives the desired answer.

8.3 − 4

Show how to sort n integers in the range 0 to $n^2 - 1$ in $O(n)$ time. Notice that the number of digits used to represent an n^2 different numbers in a k-ary number system is $d = \log_k(n^2)$. Thus considering the n^2 numbers as radix n numbers gives us that:

$$d = \log_n(n^2) = 2\log_n(n) = 2$$

Radix sort will then have a running time of $\Theta(d(n+k) = \Theta(2(n+n)) = \Theta(n)$.

8.4 − 2

Show ho to improve the worst-case running time of bucket sort to $O(n \lg n)$. Simply replace the insertion sort used to sort the linked lists with some worst case $O(n \lg n)$ sorting algorithm, e.g. merge sort. The sorting then takes time:

$$\sum_{i=0}^{n-1} O(n_i \lg n_i) \leqslant \sum_{i=0}^{n-1} O(n_i \lg n) = O(\lg n) \sum_{i=0}^{n-1} O(n_i) = O(n \lg n)$$

The total time of bucket sort is thus $O(n \lg n)$.

$9.1 - 1$

Show how to find the the second smallest element of n elements using $n + \lceil \lg n \rceil - 2$ comparisons. To find the smallest element construct a tournament as follows: Compare all the numbers in pairs. Only the smallest number of each pair is potentially the smallest of all so the problem is reduced to size $\lceil n/2 \rceil$. Continuing in this fashion until there is only one left clearly solves the problem.

Exactly $n - 1$ comparisons are needed since the tournament can be drawn as an n-leaf binary tree which has $n - 1$ internal nodes (show by induction on n). Each of these nodes correspond to a comparison.

We can use this binary tree to also locate the second smallest number. The path from the root to the smallest element (of height $\lceil \lg n \rceil$) must contain the second smallest element. Conducting a tournament among these uses $\lceil \lg n \rceil - 1$ comparisons.

The total number of comparisons are: $n - 1 + \lceil \lg n \rceil - 1 = n + \lceil \lg n \rceil - 2$.

$9.3 - 1$

Consider the analysis of the algorithm for groups of k. The number of elements less than (or greater than) the median of the medians x will be at least $\lceil \frac{k}{2} \rceil \left(\lceil \frac{1}{2} \lceil \frac{n}{k} \rceil \rceil - 2 \right) \geqslant \frac{n}{4} - k$. Hence, in the worst-case SELECT will be called recursively on at most $n - \left(\frac{n}{4} - k \right) = \frac{3n}{4} + k$ elements. The recurrence is

$$T(n) \leqslant T(\lceil n/k \rceil) + T(3n/4 + k) + O(n)$$

Solving by substitution we obtain a bound for which k the algorithm will be linear. Assume $T(n) \leqslant cn$ for all smaller n. We have:

$$\begin{aligned}
T(n) &\leqslant c \left\lceil \frac{n}{k} \right\rceil + c \left(\frac{3n}{4} + k \right) + O(n) \\
&\leqslant c(\frac{n}{k} + 1) + \frac{3cn}{k} + ck + O(n) \\
&\leqslant \frac{cn}{k} + \frac{3cn}{k} + c(k+1) + O(n) \\
&= cn \left(\frac{1}{k} + \frac{3}{4} \right) + c(k+1) + O(n) \\
&\leqslant cn
\end{aligned}$$

Where the last equation only holds for $k > 4$. Thus, we have shown that the algorithm will compute in linear time for any group size of 4 or more. In fact, the algorithm is $\Omega(n \lg n)$ for $k = 3$. This can be shown by example.

$9.3 - 3$

Quicksort can be made to run in $O(n \lg n)$ time worst-case by noticing that we can perform "perfect partitioning": Simply use the linear time select to find the median and perform the partitioning around it. This clearly achieves the bound.

$9.3 - 5$

Assume that we have a routine MEDIAN that computes the median of an array in $O(n)$ time. Show how to find an arbitrary order statistic in $O(n)$.

Algorithm 6 $\textsc{Select}(A, i)$

```
Input: Array $A$ and integer $i$.
Output: The $i$th largest element of $A$.
$x \leftarrow \textsc{Median}(A)$.
Partition $A$ around $x$
if $i \leqslant \lfloor (n+1)/2 \rfloor$ then
    Recursively find the $i$th in the first half
else
    Recursively find $(i - \lfloor (n+1)/2 \rfloor)$th in the second half
end if
```

Clearly, this algorithm does the job.

10.1 – 2

Two stacks can be implemented in a single array without overflows occuring if they grow from each end and towards the middle.

10.1 – 6

Implement a queue using two stacks. Denote the two stacks S_1 and S_2. The ENQUEUE operation is simply implemented as a push on S_1. The dequeue operation is implemented as a pop on S_2. If S_2 is empty, successively pop S_1 and push S_2. The reverses the order of S_1 onto S_2.

The worst-case running time is $O(n)$ but the amortized complexity is $O(1)$ since each element is only moved a constant number of times.

10.2 – 1

No, INSERT and DELETE can not be implemented in $O(1)$ on a linked list. A scan through the list is required for both operations. INSERT must check that no duplicates exist.

10.2 – 2

A stack can be implemented using a linked list in the following way: PUSH is done by appending a new element to the front of the list and POP is done by deleting the first element of the list.

$11.1 - 1$

Find the maximum element in a direct-address table T of length m. In the worst-case searching the entire table is needed. Thus the procedure must take $O(m)$ time.

$11.1 - 2$

Describe how to implement a dynamic set with a bitvector. The elements have no satellite data. Simply set the ith bit to 1 if the i element is inserted and set the bit to 0 if it is deleted.

$11.2 - 3$

Consider keeping the chaining lists in sorted order. Searching will still take time proportional to the length of the list and therefore the running times are the same. The only difference is the insertions which now also take time proportional to the length of the list.

$11.3 - 1$

Searching a list of length n where each element contains a long key k and a small hash value $h(k)$ can be optimized in the following way: Comparing the keys should be done first comparing the hash values and if succesfull then comparing the keys.

$12.1 - 2$

The definitions clearly differ. If the heap property allowed the elements to be printed in sorted order in time $O(n)$ we would have an $O(n)$ time comparison sorting algorithm since BUILD-HEAP takes $O(n)$ time. This, however, is impossible since we know $\Omega(n \lg n)$ is a lower bound for sorting.

$12.1 - 5$

Show that constructing a binary tree from an arbitrary list in a comparison based model must take at $\Omega(n \lg n)$ worst-case. The value of the nodes in the tree can be printed in sorted order in $O(n)$ time. The existance of an algorithm for constructing a binary tree in $o(n \lg n)$ time would thus contradict the lower bound for sorting.

$12.2 - 5$

Show that if a node in a binary search tree has two children then its successor has no left child and its predecessor has no right child. Let v be a node with two children. The nodes that immidiately precede v must be in the left subtree and the nodes that immidiately follow v must be in the right subtree. Thus the successor s must be in the right subtree and s will be the next node from v in an inorder walk. Therefore s cannot have a left child since this child since this would come before s in the inorder walk. Similarly, the predecessor has no right child.

$12.2 - 7$

Show that the inorder walk of a n-node binary search tree implemented with a call to TREE-MINIMUN followed by $n - 1$ calls to TREE-SUCCESSOR takes $O(n)$ time.

Consider the algorithm at any given node during the algorithm. The algorithm will never go to the left subtree and going up will therefore cause it to never return to this same node. Hence the algorithm only traverses each edge at most twice and therefore the running time is $O(n)$.

$12.2 - 9$

If x is a leaf node, then if $p[x] = y$ and x is the left child then running TREE-SUCCESSOR yields y. Similarly if x is the right child then running TREE-PREDECESSOR yields y.

$12.3 - 1$

Algorithm 7 TREE-INSERT(z, k)

```
Input: A node $z$ and value $k$.
Output: The binary tree with $k$ inserted.
if $z$ = nil then
   key[$z$] $\leftarrow k$
   left[$z$] $\leftarrow$ nil
   right[$z$] $\leftarrow$ nil
else
   if $k < key[z]$ then
      TREE-INSERT(left[$z$], $k$)
   else
      TREE-INSERT(right[$z$], $k$)
   end if
end if
```

12.3 – 5

The deletion operation is not commutative. A counterexample is shown in figure 1.

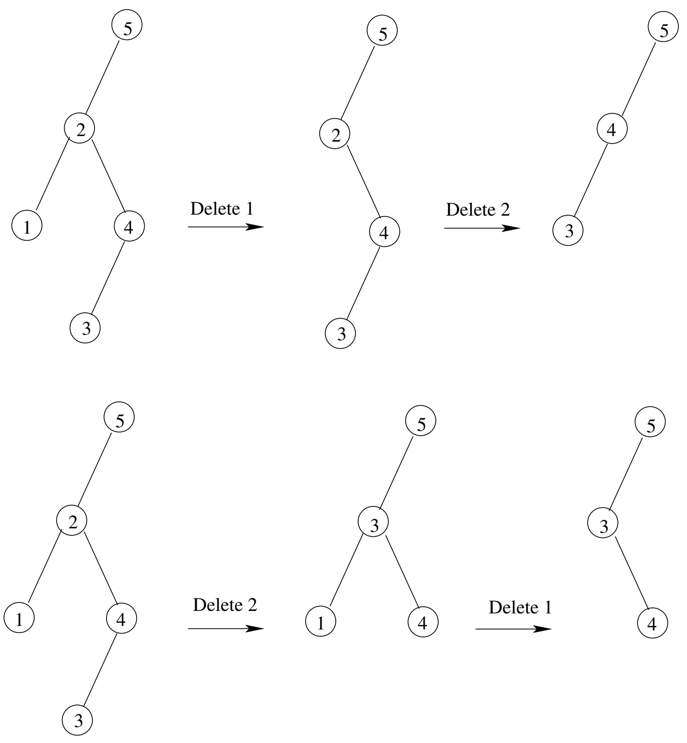

Figure 1: Two deletions where the order of the operations matter.

13.1 − 5

By property 5 the longest and shortest path must contain the same number of black nodes. By property 4 every other nodes in the longest path must be black and therefore the length is at most twice that of the shortest path.

13.1 − 6

Consider a red-black tree with black-height k. If every node is black the total number of internal nodes is $2^k - 1$. If only every other nodes is black we can construct a tree with $2^{2k} - 1$ nodes.

13.3 − 1

If we choose to set the colour of a newly inserted node to black then property 4 is not violated but clearly property 5 is violated.

13.3 − 2

Inserting the keys $41, 38, 31, 12, 19, 8$ into an initially empty red-black tree yields the trees depicted in figure 2 on page 20.

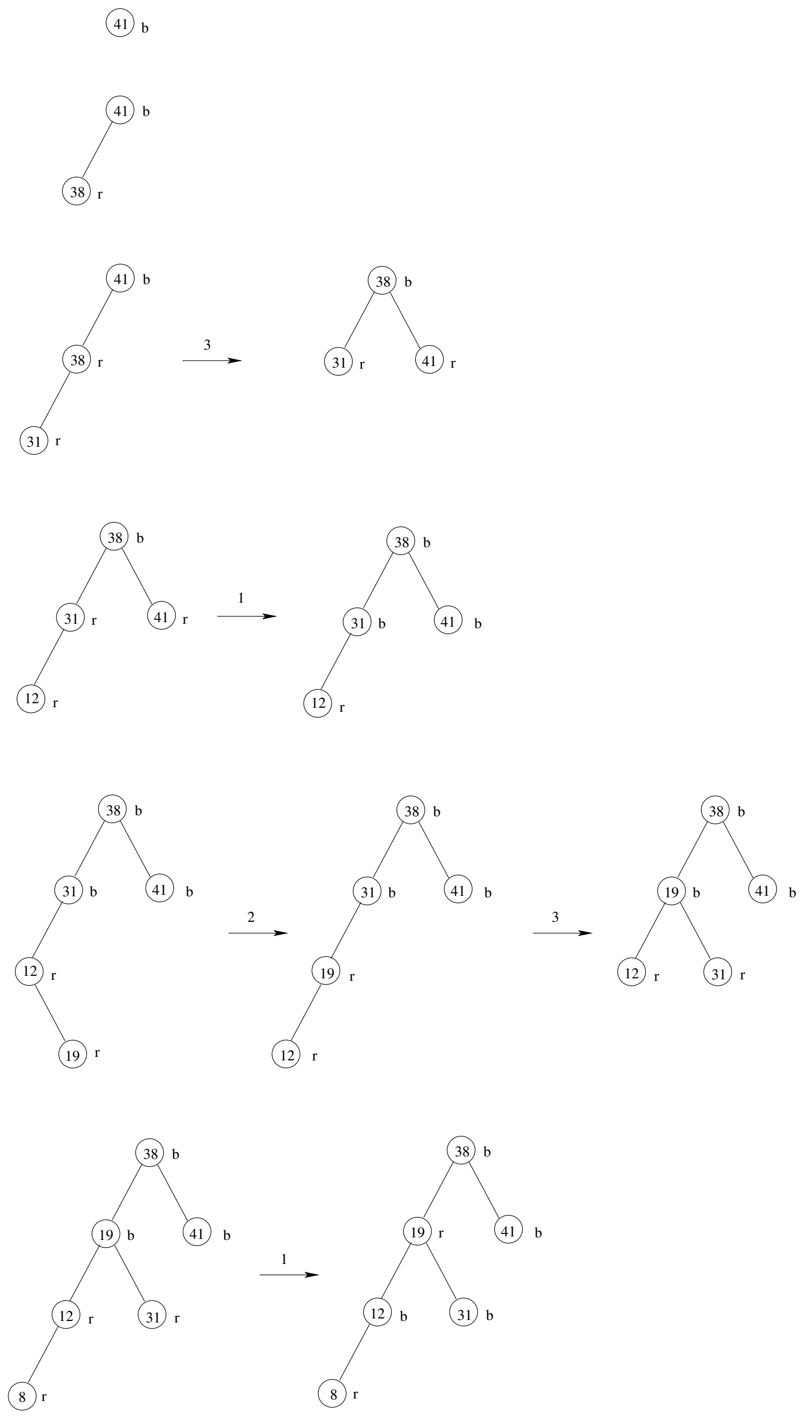

Figure 2: Inserting $41, 38, 31, 12, 19, 8$ into a red-black tree. The arrows indicate transformations. Notice that the root is always coloured black

$13.3 - 3$

Show that property 5 is preserved in figure 13.5 and 13.6 assumming the height of α, β, γ, δ and ϵ is k. For 13.5 the black height for nodes A,B and D is $k + 1$ in both cases since all the subtrees have black height k. Node C has black height $k + 1$ on the left and $k + 2$ on the right since the black height of its black children is $k+1$. For 13.6 it is clearly seen that both A, B and C have black height $k+1$. We see that the black height is well defined and the property is maintained through the transformations.

13.4 − 7

Inserting and immidiately deleting need not yield the same tree. Here is an example that alters the structure and one that changes the colour.

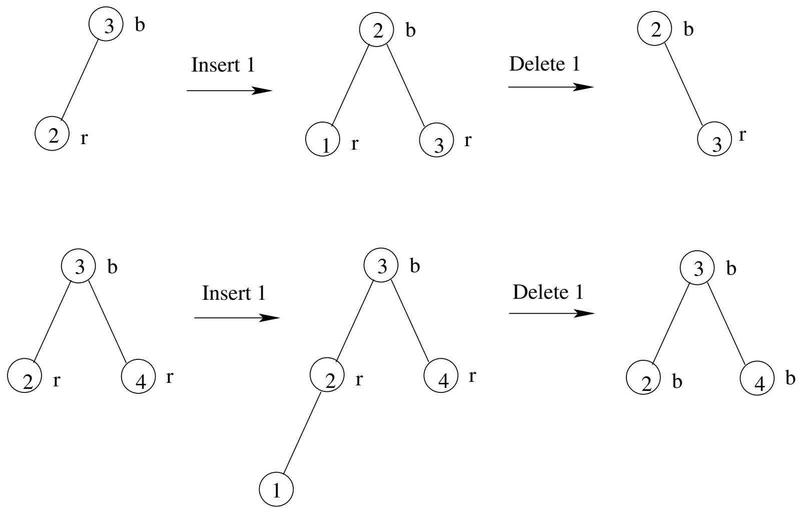

Figure 3: Inserting and deleting 1

15.1 − 5

Show that $l_1[j] = 2$ and $l_2[j] = 1$ is impossible for any j in any instance of FASTEST-WAY. Assume that this is the case and consider the values of f. We have by definition that:

$$\begin{aligned} f_1[j] &= \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_1, j) \\ f_2[j] &= \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_2, j) \end{aligned}$$

Since $l_1[j] = 2$ and $l_2[j] = 1$ we have that:

$$\begin{aligned} f_1[j-1] + a_{1,j} &> f_2[j-1] + t_{2,j-1} + a_1, j \\ f_2[j-1] + a_{2,j} &> f_1[j-1] + t_{1,j-1} + a_2, j \end{aligned}$$

By cancelling out the a's we obtain a contradiction and the statement follows.

15.2 − 1

Solve the matrix chain order for a specific problem. This can be done by computing MATRIX-CHAIN-ORDER(p) where $p = \langle 5, 10, 3, 12, 5, 50, 6 \rangle$ or simply using the equation:

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

The resulting table is the following:

i\j	1	2	3	4	5	6
1	0	150	330	405	1655	2010
2		0	360	330	2430	1950
3			0	180	930	1770
4				0	3000	1860
5					0	1500
6						0

The table is computed simply by the fact that $m[i,i] = 0$ for all i. This information is used to compute $m[i, i+1]$ for $i = 1, \dots n-1$ an so on.

15.3 − 2

Draw a nice recursion tree. The MERGESORT algorithm performs at most a single call to any pair of indices of the array that is being sorted. In other words, the subproblems do not overlap and therefore memoization will not improve the running time.

15.4 − 3

Give an efficient memoized implementation of LCS-LENGTH. This can done directly by using:

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

Algorithm 8 LCS-LENGTH(X, Y)

```
Input: The two strings X and Y.
Output: The longest common substring of X and Y.
$m \leftarrow length[X]$
$n \leftarrow length[Y]$
for $i \leftarrow 1$ to $m$ do
    for $j \leftarrow 1$ to $n$ do
        $c[i, j] \leftarrow -1$
    end for
end for
return LOOKUP-LENGTH$(X, Y, m, n)$
```

Algorithm 9 LOOKUP-LENGTH(X, Y, i, j)

```
if $c[i, j] > -1$ then
    return $c[i, j]$
end if
if $i = 0$ or $j = 0$ then
    $c[i, j] \leftarrow 0$
else
    if $x_i = y_i$ then
        $c[i, j] \leftarrow$ LOOKUP-LENGTH$(X, Y, i - 1, j - 1) + 1$
    else
        $c[i, j] \leftarrow \max\{$LOOKUP-LENGTH$(X, Y, i, j - 1)$, LOOKUP-LENGTH$(X, Y, i - 1, j)\}$
    end if
end if
return $c[i, j]$
```

$15.4 - 5$

Given a sequence $X = \langle x_1, x_2, \ldots, x_n \rangle$ we wish to find the longest monotonically increasing subsequence. First sort the elements of X and create another sequence X'. Finding the longest common subsequence of X and X' yields the longest monotonically increasing subsequence of X. The running time is $O(n^2)$ since sorting can be done in $O(n \lg n)$ and the call to LCS-LENGTH is $O(n^2)$.

$15 - 1$

Compute the bitonic tour of n points in the plane. Sort the points and enumerate from left to right: $1, \ldots n$. For any i,$1 \leqslant i \leqslant n$, and for any k,$1 \leqslant k \leqslant n$, let $B[i, k]$ denote the minimun length of two disjoint bitonic paths, one from 1 to i, the other from 1 to k. When $i = k$ we have a minumin cost bitonic tour through the first i points. When $i = k = n$ we have a minimum cost bitonic tour through all n points. Note that we need only consider disjoint paths since any non-disjoint paths cannot be optimal due to the triangle inequality. We can now describe how to compute B using dynamic programming.

First define $B[0, 0] = 0$. We will determine the value of $B[i + 1, k]$ for some fixed i and for all k, $1 \leqslant k \leqslant i + 1$ by using B-values from the first i rows and i columns of B.

Case 1: $k < i$. The minimun cost disjoint paths from 1 to $i + 1$ and from 1 to k must contain the edge $(i, i + 1)$. Therefore

$$B[i + 1, k] = B[i, k] + w(i, i + 1)$$

Case 2: $k = i$. In other words, we a looking for $B[i+1, i]$. The edge ending in $i+1$ comes from u, $1 \leqslant u < i$. Hence

$$B[i+1, i] = \min_{1 \leqslant u < 1} \{B[i, u] + w(u, i+1)\}$$

Case 3: $k = i + 1$. The two edges entering $i+1$ must come from i and from some u,$1 \leqslant u < i$. Therefore

$$B[i+1, i+1] = \min_{1 \leqslant u < 1} \{B[i, u] + w(i, i+1) + w(u, i+1)\}$$

$15 - 4$

The problem can be transformed into a tree colouring problem where we consider the supervisor tree and colour each node red if the employee is attending and white otherwise. The parent of a red node must be white. We wish to colour the tree so that the sum of the conviality of the nodes is maximised. Let $\nu = T(x, c)$ be the conviviality of the three rooted at the node x that is coloured with colour c. We can construct the following recursion:

If x is a leaf with convivialty ν and colour c then:

$$T(x, c) = \begin{cases} \nu & \text{if } x = \textsc{Red} \\ 0 & \text{if } x = \textsc{White} \end{cases}$$

If x is not a leaf then similarly:

$$T(x, c) = \begin{cases} \nu + \sum_i T(x.\mathit{child}_i, \textsc{White}) & \text{if } x = \textsc{Red} \\ \sum_i \max(T(x.\mathit{child}_i, \textsc{White}), T(x.\mathit{child}_i, \textsc{Red})) & \text{if } x = \textsc{White} \end{cases}$$

The maximal conviviality $\nu_{\max}$ is then given by $\nu_{\max} = \max(T(\mathit{root}, \textsc{White}), T(\mathit{root}, \textsc{Red})$. Implementing this recursion yields a straight forward algorithm using memoization. Since there is exactly one subproblem for each node the running time will be $O(n)$ for an n node tree.

Notes for the exercises

- Thanks to Jarl Friis for the tedious calculation of the table in exercise $15.2 - 1$.
- Thanks to Pawel Winter for providing a solution to $15 - 1$.

16.1 − 3

Find the smallest number of lecture halls to schedule a set of activities S in. To do this efficiently move through the activities according to starting and finishing times. Maintain two lists of lecture halls: Halls that are busy at time t and halls that are free at time t. When t is the starting time for some activity schedule this activity to a free lecture hall and move the hall to the busy list. Similarly, move the hall to the free list when the activity stops. Initially start with zero halls. If there are no halls in the free list create a new hall.

The above algorithm uses the fewest number of halls possible: Assume the algorithm used m halls. Consider some activity a that was the first scheduled activity in lecture hall m. i was put in the mth hall because all of the $m - 1$ halls were busy, that is, at the time a is scheduled there are m activities occuring simultaneously. Any algorithm must therefore use at least m halls, and the algorithm is thus optimal.

The algorithm can be implemented by sorting the activities. At each start or finish time we can schedule the activities and move the halls between the lists in constant time. The total time is thus dominated by sorting and is therefore $\Theta(n \lg n)$.

16.1 − 4

Show that selecting the activity with the least duration or with minimun overlap or earliest starting time does not yield an optimal solution for the activity-selection problem. Consider figure 4:

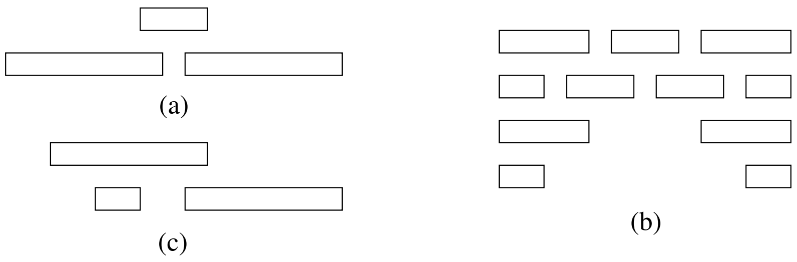

Figure 4: Three examples with other greedy strategies that go wrong

Selecting the activity with the least duration from example a will result in selecting the topmost activity and none other. Clearly, this is worse than the optimal solution obtained by selecting the two activities in the second row.

The activity with the minimun overlap in example b is the middle activity in the top row. However, selecting this activity eliminates the possibility of selecting the optimal solution depicted in the second row.

Selecting the activity with the earliest starting time in example c will yield only the one activity in the top row.

16.2 − 2

The 0/1 knapsack problem exibits the optimal substructure given in the book: Let i be the highest numbered item among $1, \dots, n$ items in an optimal solution S for W with value $v(S)$. Then $S' = S - \{i\}$ is an optimal solution for $W - w_i$ with value $v(S') = v(S) - v_i$.

We can express this in the following recursion. Let $c[i, w]$ denote the value of the solution for items $1, \dots, i$ and maximum weight w.

$$c[i, w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c[i-1, w] & \text{if } w_i > w \\ \max(v_i + c[i-1, w - w_i], c[i-1, w]) & \text{if } i > 0 \text{ and } w \geqslant w_i \end{cases}$$

Notice that the last case determines whether or not the ith element should be included in an optimal solution. We can use this recursion to create a straight forward dynamic programming algorithm:

Algorithm 10 DYNAMIC-0-1-KNAPSACK (v, w, n, W)

```
Input: Two sequences $v = \langle v_1, \dots, v_n \rangle$ and $w = \langle w_1, \dots, w_n \rangle$ the number of items $n$ and the maximum weight $W$.
Output: The optimal value of the knapsack.
for $w \leftarrow 0$ to $W$ do
  $c[0, w] \leftarrow 0$
end for
for $i \leftarrow 1$ to $n$ do
  $c[i, 0] \leftarrow 0$
  for $w \leftarrow 1$ to $W$ do
    if $w_i \leqslant w$ then
      if $v_i + c[i-1, w-w_i] > c[i-1, w]$ then
        $c[i, w] \leftarrow v_i + c[i-1, w-w_i]$
      else
        $c[i, w] \leftarrow c[i-1, w]$
      end if
    else
      $c[i, w] \leftarrow c[i-1, w]$
    end if
  end for
end for
return $c[n, W]$
```

For the analysis notice that there are $(n+1) \cdot (W+1) = \Theta(nW)$ entries in the table c each taking $\Theta(1)$ to fill out. The total running time is thus $\Theta(nW)$.

$16.2 - 5$

Describe an algorithm to find the smallest unit-length set, that contains all of the points $\{x_1, \dots x_n\}$ on the real line. Consider the following very simple algorithm: Sort the points obtaining a new array $\{y_1, \dots y_n\}$. The first interval is given by $[y_1, y_1 + 1]$. If y_i is the leftmost point not contained in any existing interval the next interval is $[y_i, y_i + 1]$ and so on.

This greedy algorithm does the job since the rightmost element of the set must be contained in an interval and we can do no better than the interval $[y_1, y_1 + 1]$. Additionally, any subproblem to the optimal solution must be optimal. This is easily seen by considering the problem for the points greater than $y_1 + 1$ and arguing inductively.

$16.3 - 8$

Show that we cannot expect to compress a file of randomly chosen bits. Notice that the number of possible source files S using n bits and compressed files E using n bits is $2^{n+1} - 1$. Since any compression algorithm must assign each element $s \in S$ to a distinct element $e \in E$ the algorithm cannot hope to actually compress the source file.

17.1 − 2

If a DECREMENT operation is added we can easily force the counter to change all bits per operation by calling DECREMENT and INCREMENT on 2^{k-1}. This gives a total running time of $\Theta(nk)$.

17.1 − 3

Consider a datastructure where n operations are performed. The ith operation costs i if i is an exact power of two and 1 otherwise. Determine the amortized cost of each operation using the aggregate method. Let c_i denote the cost of the ith operation. Summing the cost of the n operations yield:

$$\sum_{i=0}^{n} c_i \leqslant n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j = n + 2^{\lfloor \lg n \rfloor + 1} - 1 < n + 2n - 1 < 3n$$

Thus the amortized time of each operation is less than $3n/n = O(1)$.

17.2 − 1

We wish to show an $O(1)$ amortized cost on stack operations where the stack is modified such that after k operations a backup is made. The size of the stack never exceeds k and if we assign an extra credit to each stack operation we can always pay for copying.

17.2 − 2

Redo exercise 18.1 − 3 using the accounting method. Charging 3 credits per operation will do the job. This can be seen by example and by exercise 18.1 − 3.

17.2 − 3

Keeping a pointer to the highest order bit and charging one extra credit for each bit we set allows us to do RESET in amortized $O(1)$ time.

17.3 − 2

Redo exercise 18.3 − 2 using the potential method. Consider operation number $i = 2^j + k$, where j and $k \geqslant 0$ are integers and j is chosen as large as possible. Let the potential function be given by $\Phi(D_i) = 2k$. Clearly, this function satifies the requirements. There are two cases to consider for the ith operation:

If $k = 0$ then the actual cost is i and the amortized cost is given by:

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= i + 0 - 2 \cdot (2^j - 1 - 2^{j-1}) \\ &= i - (2 \cdot (2^j - 2^{j-1}) - 2) \\ &= i - (2^j \cdot (2 - 1) - 2) \\ &= i - i + 2 \\ &= 2\end{aligned}$$

Otherwise the actual cost will be 1 and we find that

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + 2k - 2(k-1) \\ &= 3\end{aligned}$$

17.4 − 3

Show that the amortized cost of TABLE-DELETE when $\alpha_{i-1} \geqslant 1/2$ is bounded above by a constant. Notice that the ith operation cannot cause the table to contract since contract only occurs when $a_i < 1/4$. We need to consider cases for the load factor α_i. For both cases $num_i = num_{i-1} - 1$ and $size_i = size_{i-1}$.

Assume $\alpha_i \geqslant 1/2$.

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= 1 + (2 \cdot num_i - size_i) - (2 \cdot (num_i + 1) - size_i) \\ &= -1\end{aligned}$$

Then consider $\alpha_i < 1/2$.

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot num_i - size_i) - (size_{i-1}/2 - num_{i-1}) \\ &= 3 \cdot num_i - 3/2 \cdot \mathrm{size}_i + 2 \\ &= 3\alpha_i \cdot size_i - 3/2 \cdot \mathrm{size}_i + 2 \\ &< 3/2 \cdot size_i - 3/2 \cdot \mathrm{size}_i + 2 \\ &= 2\end{aligned}$$

17 − 2

Construct a dynamic binary search data structure. Let $k = \lceil \lg(n+1) \rceil$. Maintain k sorted arrays $A_0, A_1, \ldots A_{k-1}$ such that the length of A_i is 2^i.

a. A SEARCH operation can be implemented by using a binary search on all the arrays. The worst-case complexity of this must be:

$$\sum_{i=0}^{k} \lg(2^i) = \sum_{i=0}^{k} i = \frac{k(k+1)}{2} = O(k^2) = O(\lg^2 n)$$

b. The worst-case of the INSERT operation occurs when all the sorted array have to be merged into a new array. That is, when n increases from $2^k - 1$ to 2^k for some k. Using k-way merging as in exercise 6.5 − 8 with a total of n elements yields a running time of $O(n \lg k) = O(n \lg\lg n)$.

From the analysis of incrementing a binary counter we have that the ith bit is flipped a total of $\lfloor n/2^i \rfloor$ times in n INCREMENT operations. The correspondance to this problem is that every time the ith bit in n is flipped we need to merge the lists $A_i, A_{i-1}, \ldots A_0$ using time $O(2^i \lg i)$. The total running time is thus:

$$\sum_{i=1}^{\lfloor \lg n \rfloor} \lfloor n/2^i \rfloor 2^i \lg i < n \sum_{i=1}^{\lg n} \lg i = n \lg(\prod_{i=1}^{\lg n} i) = O(n \lg((\lg n)!)) = O(n \lg n \lg\lg n)$$

The amortized complexity is therefore $\lg n \lg\lg n$.

c. The DELETE operation should be implemented like the one used in dynamic tables (section 17.4). One should wait linear time before deallocating an array in order to avoid the worst-case complexity.

$19.1-1$

If x is a node in a binomial heap how does *degree*[x] compare to *degree*[*sibling*[x]]? Assume x is the root of a subtree B_k with degree k and x indeed has a sibling. If x is not a root then, by definition, the sibling must be root of a subtree B_{k-1} with degree $k-1$. If x is a root then the next tree occuring in the root list must be the tree B_i corresponding to the index i of the next 1 bit in the binary representation of n.

$19.1-2$

If x is a nonroot node in a binomial tree *degree*[p[x]] will be *degree*[x] $+ 1$. This is clear from the definition.

$19.2-7$

Uniting two binomial heaps of sizes n_1 and n_2 results in a heap of size $n = n_1 + n_2$. Notice that if the ith bit in n is 1 the resulting heap will contain the binomial tree B_i. This clearly corresponds to binary addition and binary incrementation.

$19-2$

The algorithm MST-MERGEABLE-HEAP(G) is trivially implemented using the binomial heap operations. Consider the running of the algorithm step by step:

- The **for** loop of line $2-4$ requires $O(V)$ MAKE-HEAP operations and a total of $O(E)$ INSERT operations in line 4. Note that the size of each E_i set can be at most $O(V)$ by definition. The total time is thus $O(V + E \lg V)$.
- Consider the **while** loop of line $5-12$.
  - We can at most extract $O(E)$ edges in line 7 taking a total of $(E \lg V)$ time.
  - The $i \neq j$ check can be done in time $O(1)$ by enumerating the sets.
  - The **then** branch is at most taken $O(V)$ times since it reduces the number of V_i's by one every time. Insertion into T can be done in $O(1)$ time using a linked list. Merging V_i's take $O(\lg V)$. Merging E_i's take $O(\lg E) = O(\lg V^2) = O(\lg V)$.
- The total time of the **while** loop is then $O(E \lg V + V \lg V + V \lg V)$.

The overall running time is seen to be $O(E \lg V)$.

$20.2 - 5$

If all the comparison based mergeable-heap operations ran in time $O(1)$ then we could sort n numbers in time $O(n)$ using INSERT and EXTRACT-MIN. This contradicts the lower bound of $\Omega(n \lg n)$.

21.3 − 2

Give a non-recursive implementation of FIND-SET with path compression. The following algorithm does the job using pointer reversal:

Algorithm 11 FIND-SET(x)

```
Input: The node $x$.
Output: The root of the tree *root* to the representative of $x$.
$y \leftarrow p[x]$
$p[x] \leftarrow x$
while $x \neq y$ do
    $z \leftarrow p[y]$
    $p[y] \leftarrow x$
    $x \leftarrow y$
    $y \leftarrow z$
end while
root $\leftarrow x$
$y \leftarrow p[x]$
$p[x] \leftarrow x$
while $x \neq y$ do
    $z \leftarrow p[y]$
    $p[y] \leftarrow$ root
    $x \leftarrow y$
    $y \leftarrow z$
end while
return root
```

The first while loop traverses the path from x to the root of the tree while reversing all parent pointers along the way. The second while loop returns along this path and sets all parent pointers along the way to point to the root.

21.3 − 3

Construct a sequence of m MAKE-SET, FIND-SET and UNION operations, n of which are MAKE-SET, that takes $\Omega(m \lg n)$ time when we use union by rank only.

First perform the n (assume n is a power of 2 for simplicity) MAKE-SET operations on each of the elements $\{x_1, x_2, \dots x_3\}$. Then perform a union on each pair (x_1, x_2), (x_3, x_4) and so on yielding $n/2$ new sets. Continue the process until there is only a single set left. This uses a total of $n - 1$ operations and produces a tree with depth $\Omega(\lg n)$. Perform additional $k \geqslant n$ FIND-SET operations on the node of greatest depth. Setting $m = n + n - 1 + k$ gives the desired running time of $\Omega(n + m \lg n) = \Omega(m \lg n)$.

21.4 − 4

Show that the running time of the disjoint-set forest with union by rank only is $O(m \lg n)$. First notice that the rank of a node is at most the height of the subtree rooted at that node. By exercise 21.4 − 2 every node has rank at most $\lfloor \lg n \rfloor$ and therefore the height h of any subtree is at most $\lfloor \lg n \rfloor$. Clearly, each call to FIND-SET (and thus UNION) takes at most $O(h) = O(\lg n)$ time.

21 − 1

Consider an off-line minimum problem, where we are given n INSERT and m EXTRACT-MIN calls. The elements are all from the set $\{1, \dots, n\}$. We are required to return an array *extracted*$[1 \dots m]$ such that *extracted*$[i]$ is the return value of the ith EXTRACT-MIN call.

a. Consider the following sequence:

$4, 8, E, 3, E, 9, 2, 6, E, E, E, 1, 7, E, 5$

The corresponding *extracted* array is:

$4, 3, 2, 6, 8, 1$

b. Running an example convinces one of the correctness of the algorithm.

c. Using UNION-FIND techniques we can construct an efficient implementation of the algorithm. Initially create disjoint-sets for the subsequences $I_1, \ldots I_{m+1}$ and place the representative of each set in a linked list in sorted order. Additionally, label each representative with its subsequence number. We proceed as follows:

- Line 2 is implemented by a FIND-SET operation on i yielding the representative of subsequence I_j labeled j.
- In line 5 the next set can found from the root as the next set in the linked list.
- Line 6 is implemented with a UNION operation and a deletion in the linked list.

The overall running time is seen to be $O(m\alpha(n))$.

$21 - 2$

Consider the depth-determination problem where we maintain a forest $F = \{T_i\}$ of rooted trees and support the MAKE-TREE, FIND-DEPTH and Graft operations.

a. Show that m operations using a simple disjoint trees takes $\Theta(m^2)$. Create single node trees $v_0, \ldots, v_k$ with MAKE-TREE, where $k = \frac{1}{3}m$. Then use $k-1$ GRAFT operations to link them into a single path $v_0, \ldots, v_k$ with root v_k. Finally, call FIND-DEPTH(v_0) k times. These $m+1$ operations takes $(k+1) + k + k^2 < (\frac{1}{3}m)^2 = \Theta(m^2)$ time.

b. implement MAKE-DEPTH(v) by creating a disjoint-set S with the node v an setting $d[v] \leftarrow 0$.

c. Consider FIND-DEPTH(v) and let $v = v_0, \ldots, v_k$ be the path to the root. For each v_i compute $d[v_i] \leftarrow \sum_{i=j}^{k-1} d[v_j]$. Then use the ordinary FIND-SET with path compression.

d. Show how to implement GRAFT(r, v). We will use the ordinary UNION operation and update the pseudodistances appropriately. There are two cases to consider as shown in figure 5.

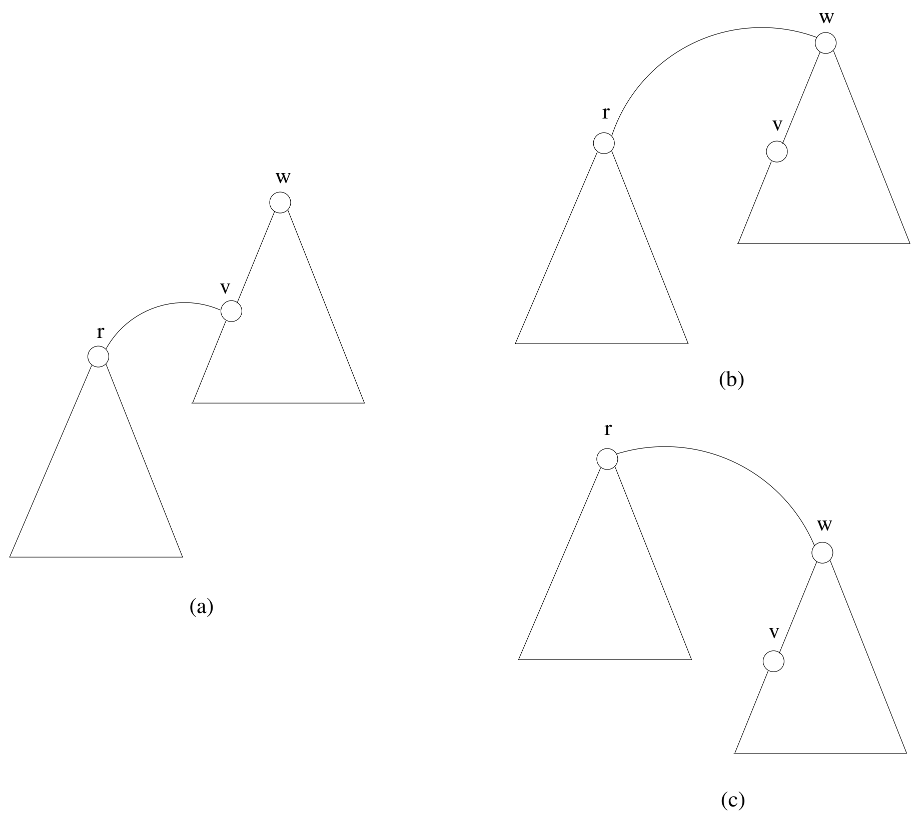

Figure 5: (a) The tree before GRAFT(r, v). The cases on the right show the tree after GRAFT(r, v) and the two cases: (b) $rank(r) \leqslant rank(w)$ (c) $rank(r) > rank(w)$

Let the path from v to w be $v = v_0, \ldots, v_k = w$. The depth of any node in the subtree rooted at r is increased by $\sum_{i=0}^{k} d[v_i]$ while the depths of the other nodes is unaffected. For case (b) this can be achieved by setting $d[r] \leftarrow d[r] + \sum_{i=0}^{k-1} d[v_i]$. For case (c) set $d[r] \leftarrow d[r] + \sum_{i=0}^{k} d[v_i]$ and then set $d[w] \leftarrow d[w] - d[r]$. Both cases are easily handled without extra cost to the UNION operation.

e. Clearly, the total cost is $O(m\alpha(n))$ as in UNION-FIND algorithm.

22.1 − 1

Given the adjacency-list representation of a graph the out and in-degree of every node can easily be computed in $O(E + V)$ time.

22.1 − 3

The transpose of a directed graph G^T can be computed as follows: If the graph is represented by an adjacency-matrix A simply compute the transpose of the matrix A^T in time $O(V^2)$. If the graph is represented by an adjacency-list a single scan through this list is sufficient to construct the transpose. The time used is $O(E + V)$.

22.1 − 5

The square of a graph can be computed as follows: If the graph is represented as an adjacency-matrix A simply compute the product A^2 where multiplication and addition is exchanged by and'ing and or'ing. Using the trivial algorithm yields a running time of $O(n^3)$. Using Strassens algorithm improves the bound to $O(n^{\lg 7}) = O(n^{2,81})$.

If we are using the adjacency-list representation we can simply append lists eliminating duplicates. Assuming the lists are sorted we can proceed as follows: For each node v in some list replace the v with $Adj[v]$ and merge this into the list. Each list can be at most V long and therefore each merge operation takes at most $O(V)$ time. Thus the total running time is $O((E+V)V) = O(E+V^2)$.

22.1 − 6

Notice that if edge is present between vertices v and u then v cannot be a sink and if the edge is not present then u cannot be a sink. Searching the adjancency-list in a linear fashion enables us to exclude one vertex at a time.

22.2 − 3

If breadth-first-search is run on a graph represented by an adjacency-matrix the time used scanning for neighbour can increase to $O(V^2)$ yielding a total running time of $O(V^2 + V)$.

22.2 − 6

The problem is the same as determining if a graph is bipartite. From graph theory we know that this is so if and only if every cycle has even length. Using this fact we can simply modify breadth-first-search to compare the current distance with the distance of an encountered gray vertex. Additionally, every other vertex within a cycle will be in one of the vertex sets and we can therefore easily construct the partition. The running time is $O(E + V)$.

22.2 − 7

The diameter of a tree can computed in a bottom-up fashion using a recursive solution. If x is a node with a depth $d(x)$ in the tree then the diameter $D(x)$ must be:

$$D(x) = \begin{cases} \max\{\max_i\{D(x.\mathit{child}_i)\}, \max_{ij}\{d(x.\mathit{child}_i) + d(x.\mathit{child}_j)\} + 2\}, & \text{if } x \text{ is an internal node} \\ 0 & \text{if } x \text{ is a leaf} \end{cases}$$

Since the diameter must be in one of the subtrees or pass through the root and the longest path from the root must be the depth. The depth can easily be computed at the same time. Using dynamic programming we obtain a linear solution.

Actually, the problem can also be solved by computing the longest shortest path from an arbitrary node. The node farthest away will be the endpoint of a diameter and we can thus compute the longest shortest path from this node to obtain the diameter. See relevant litterature for a proof of this.

22.3 − 1

In the following table we indicate if there can be an edge (i, j) with the specified colours during a depth-first search. If the entry in the table is present we also indicate which type the edge might be: **T**ree, **F**orward or **B**ack edge. For the directed case:

(i, j)	WHITE	GRAY	BLACK
WHITE	TBFC	BC	C
GRAY	TF	TFB	TFC
BLACK		B	TFBC

For the undirected case:

(i, j)	WHITE	GRAY	BLACK
WHITE	TB	TB	
GRAY	TB	TB	TB
BLACK		TB	TB

22.4 − 3

Show how to determine if an undirected graph contains a cycle in $O(V)$ time. A depth-first search on an undirected graph yields only tree edges and back edges as shown in the table in exercise 22.3 − 1. Observe that an undirected graph is acyclic if and only if a depth-first search yields no back edges.

We now perform a depth-first search and if we discover a back edge then the graph must be acyclic. The time taken is $O(V)$ since if we discover more than V distinct edges the graph must be acyclic and we will have seen a back edge.

22 − 3

a. If G has an Euler tour any path going "into" a vertex must "leave" it. Conversely, if the in and out-degrees of any vertex is the same any we can construct a path that visits all edges.

b. Since the edges of any Euler graph can be split into disjoint cycles, we can simply find these cycles and "merge" them into an Euler tour.

Notes for the exercises

- Thanks to Stephen Alstrup for providing a solution to exercise 22.2 − 7.

$23.1-1$

Show that a minimum-weight edge (u,v) belongs to some minimum spanning tree. If (u,v) is a minimum weight edge then it will be safe for any cut with u and v on separate sides. The statement follows.

$23.1-3$

Show that if an edge (u,v) belongs to some minimum spanning tree it is a light edge of some cut. Consider a cut with u and v on separate sides. If (u,v) is not a light edge then clearly the graph would not be a minimum spanning tree.

$23.1-5$

Show that if e be the maximum-weight edge on some cycle in $G=(V,E)$ then e is not part of a minimum spanning tree of G. If e was in the minimum spanning tree then replacing it with any other edge on the cycle would yield a "better" minimum spanning tree.

$23.2-1$

Given a minimum spanning tree T we wish to sort the edges in Kruskal's algorithm such that it produces T. For each edge e in T simply make sure that it preceeds any other edge not in T with weight $w(e)$.

$23.2-3$

Consider the running times of Prims algorithm implemented with either a binary heap or a Fibonnacci heap. Suppose $|E|=\Theta(V)$ then the running times are:

- Binary: $O(E\lg V)=O(V\lg V)$
- Fibonnacci: $O(E+V\lg V)=O(V\lg V)$

If $|E|=\Theta(V^2)$ then:

- Binary: $O(E\lg V)=O(V^2\lg V)$
- Fibonnacci: $O(E+V\lg V)=O(V^2)$

The Fibonnacci heap beats the binary heap implementation of Prims algorithm when $|E|=\omega(V)$ since $O(E+V\lg V)=O(V\lg V)$ t for $|E|=O(V\lg V)$ but $O(E\lg V)=\omega(V\lg V)$ for $|E|=\omega(V)$. For $|E|=\omega(V\lg V)$ the Fibonnacci version clearly has a better running time than the ordinary version.

$23.2-4$

Assume that $E\geqslant V-1$. The running time of Kruskals algorithm can be analysed as follows:

- Sorting the edges: $O(E\lg E)$ time.
- $O(E)$ operations on a disjoint-set forest taking $O(E\alpha(V))$.

The sort dominates and hence the total time is $O(E\lg E)$. Sorting using counting sort when the edges fall in the range $1,\ldots,|V|$ yields $O(V+E)=O(E)$ time sorting. The total time is then $O(E\alpha(V))$. If the edges fall in the range $1,\ldots,W$ for any constant W we still need to use $\Omega(E)$ time for sorting and the total running time cannot be improved further.

$23.2 - 5$

The running time of Prims algorithm is composed :

- $O(V)$ initialization.
- $O(V \cdot \text{time for } \textsc{Extract-Min})$.
- $O(E \cdot \text{time for } \textsc{Decrease-Key})$.

If the edges are in the range $1, \ldots, |V|$ the Van Emde Boas priority queue can speed up EXTRACT-MIN and DECREASE-KEY to $O(\lg \lg V)$ thus yielding a total running time of $O(V \lg \lg V + E \lg \lg V) = O(E \lg \lg V)$. If the edges are in the range from 1 to W we can implement the queue as an array $[1 \ldots W + 1]$ where the ith slot holds a doubly linked list of the edges with weight i. The $W + 1$st slot contains ∞. EXTRACT-MIN now runs in $O(W) = O(1)$ time since we can simply scan for the first nonempty slot and return the first element of that list. DECREASE-KEY runs in $O(1)$ time as well since it can be implemented by moving an element from one slot to another.

$23 - 1$

Let $G = (V, E)$ be an undirected graph with nonnegative distinct edge weights. We will use the following property of minimum spanning trees: If u and v are not connected in some minimum spanning tree T then the weight of (u, v) must be greater or equal than the weight of any edge on the path from u to v in T. Otherwise we could replace (u, v) with a heavier edge on the path to obtain a lighter minimum spanning tree.

a. Show that the minimum spanning tree is unique but the second-best minimum spanning tree need not be unique. Assume T and T' are distinct minimum spanning trees. Then some edge (u, v) is in T but not in T'. Since all edge weights are distinct the edges on the unique path from u to v in T' must then be strictly lighter than (u, v) contradicting the fact that T is a minimum spanning tree.

It can easily be seen by example that the second-best minimum spanning trees is not unique.

b. Show that a second-best minimum spanning tree can be obtained from the minimum spanning tree by replacing a single edge from the tree with another edge not in the tree.

Let T be a minimun spanning tree. We wish to find a tree that has the smallest possible weight that is larger than the weight of T. We can insert an edge $(u, v) \notin T$ by removing some other edge on the unique path between u and v. By the above property such a replacement must increase the weight of the tree. By carefully considering the cases it is seen that replacing two or more edges will not produce a tree better than the second best minimum spanning tree.

c. Let $\max(u, v)$ be the weight of the edge of maximum weight on the unique path between u and v in a spanning tree. To compute this in $O(V^2)$ time for all nodes in the tree do the following:

For each node v perform a traversal of the tree. Inorder to compute $\max(v, k)$ for all $k \in V$ simply maintain the largest weight encounted so far for the path being investigated. Doing this yields a linear time algorithm for each node and we therefore obtain a total of $O(V^2)$ time.

d. Using the idea of the second subexercise and the provided algorithm in the third subexercise, we can now compute T_2 from T in the following way:

Compute $\max(u, v)$ for all vertices in T. Compute for any edge (u, v) not in T the difference $w(u, v) - \max(u, v)$. The two edges yielding the smallest positive difference should be replaced.

$23-4$

We consider the three proposed minimum spanning tree algorithms. For each we prove or disprove it's correctness and give an efficient implementation.

a. The algorithm clearly produces a spanning tree T. Consider two vertices u and v such that the edge (u,v) is in G but not in T. Then $w(u,v)$ is at least as large as any weight of an edge e on the path from u to v since otherwise e would have been removed earlier. By exercise $23.1-5$ this implies that the algorithm produces a minimum spanning tree.

The sort on the edges can be done in $O(E \lg E) = O(E \lg V)$ time. Using a depth-first search to determine connectivity in line 5 the total running time is $O(E \lg V + E(V+E)) = O(E^2 + EV)$. The running time can be reduced using results on the "decremental connectivity problem".

b. This algorithm simply produces a spanning tree, but clearly does not gaurantee that the tree is of minimum weight. Line 3 and 4 in the algorithm can be implemented using disjoint-set operations yielding a total running time of $O(E\alpha(V))$.

c. The algorithm produces a spanning tree T. Observe that on completion any edge (u,v) in G but not in T will be a maximum weight edge on the path from u to v. Again by exercise $23.1-5$ the algorithm produces a minimum spanning tree.

We can implement the algorithm as follows:

- Line 3 can be done in $O(1)$ time by appending the edge to the proper adjancency list.
- Line 4 can be done in $O(V)$ time using a depth-first search as described in exercise $22.4-3$.
- Since the number of edges in T is at most $O(V)$ line 5 can be done using a linear search on the edges in $O(V)$ time.
- Line 6 can be done using a search on the adjancency list taking at most $O(V)$ time.

The **for** loop iterates at most $O(E)$ times and the total running time is thus $O(EV)$.

24.1 − 3

Let m be the maximum over all pairs of vertices $u, v \in V$ of the minimum number of edges in a shortest path from u to v. Stopping after $m + 1$ iterations in BELLMAN-FORD line 2 will produce shortest paths.

24.2 − 4

We count the number of directed paths in a directed acyclic graph $G = (V, E)$ as follows. First perform a topological sort of the input. Then for all $v \in V$ compute, $B(v)$ defined as follows.

$$B(v) = \begin{cases} 1 & v \text{ is last in the order} \\ 1 + \sum_{(v,w)\in E} B(w) & \text{otherwise} \end{cases}$$

$B(v)$ computes the number of directed paths beginning at v since if v is last in the order the only path starting at v is the empty one. Otherwise for each node w, $(v, w) \in E$, (v, w) concatenated with the paths from w and the empty path are the paths starting from v. We then compute the number of directed paths after v in the topological order. We denote this by $D(v)$ and we obtain the following.

$$D(v) = B(v) + \sum_{(v,w)\in E} D(w)$$

Since the nodes of G are ordered topologically $B(v)$ and $D(v)$ can be computed in linear. Thus the total running time is $O(E + V)$.

24.3 − 3

Consider stopping Dijkstra's algorithm just before extracting the last vertex v from the priority-queue. The shortest path estimate of this vertex must the shortest path since all edges going into v must have been relaxed. Additionally, v was to be extracted last so it will have the largest shortest path of all vertices and any relaxation from v will therefore not alter shortest path estimates. Therefore the modified algorithm is correct.

24.3 − 4

Consider to problem of computing the most reliable channel between two vertices. Observe that this is equivalent to a shortest path problem on the graph with $w(e) = \lg(r(e))$ for all $e \in E$ which can be solved using Dijkstra's algorithm. The reliability of the most reliable path to any node v can then be found as $2^{d[v]}$.

24.3 − 6

Consider running Dijkstra's algorithm on a graph, where the weight function is $w : E \to \{1, \dots W - 1\}$. To solve this efficiently, implement the priority queue by an array A of length $WV + 1$. Any node with shortests path estimate d is kept in a linked list at $A[d]$. $A[WV + 1]$ contains the nodes with ∞ as estimate.

EXTRACT-MIN is implemented by searching from the previous minimun shortest path estimate until a new is found. DECREASE-KEY simply moves vertices in the array. The EXTRACT-MIN operations takes a total of $O(VW)$ and the DECREASE-KEY operations take $O(E)$ time in total. Hence the running time of the modified algorithm will be $O(VW + E)$.

24.3 − 7

Consider the problem from the above exercise. Notice that every time a node v is extracted by EXTRACT-MIN the relaxations performed on the neighbour of v gives shortests path estimates in the range $\{d[v], \dots d[v] + W - 1\}$. Hence after every EXTRACT-MIN operation only W distinct shortest path estimates are in the priority queue at any time.

Converting the array implementation to a binary heap of the previous exercise must give a running time of $O(V + E) \lg W)$ since both the EXTRACT-MIN operation and the DECREASE-KEY operation take $O(\lg W)$ time. If we use a fibonnacci heap the running time can be further improved to $O(V \lg W + E)$.

24 − 2

We consider d-dimensional boxes.

a. The nesting relation is clearly transitive.

b. We can determine if a box nests within another by sorting the dimensions of each box and comparing them sequentially.

c. To find the longest nest sequence we determine the nesting relations on all boxes by sorting each in $O(d \lg d)$ time and comparing then pairwise in $O(d)$ time for each of the $O(n^2)$ pairs. This produces a partial relation (by definition of nesting) and thus a directed acyclic graph in which we find the longest path using $O(n^2+n)$ time. The total running time is thus $O(d \lg d + dn^2 + n^2 + n) = O(d(n^2 + \lg d))$.

$25.1-3$

The identity matrix for "multiplication" should look as the one given in the exercise since 0 is the identity for $+$ and ∞ is the identity for min.

$25.1-8$

By overriding previous matrices we can reduce the space used by FASTER-ALL-PAIRS-SHORTEST-PATH to $\Theta(n^2)$.

$25.1-9$

The presence of a negative-weight cycle can be determined by looking at the diagonal of the matrix $L^{(n-1)}$ computed by an all-pairs shortest-path algorithm. If the diagonal contains any negative number there must be a negative-weight cycle.

$25.1-10$

As in the previous exercise when can determine the presence of a negative-weight cycle by looking for a negative number in the diagonal. If $L^{(m)}$ is the first time for which this occurs then clearly the negative-weight cycle has length m. We can either use SLOW-ALL-PAIRS-SHORTEST-PATH in the straightforward manner or perform a binary search for m using FASTER-ALL-PAIRS-SHORTEST-PATH.

$25.2-4$

As in exercise $25.1-8$ overriding the result of previous calculations does not alter the correctness of the algorithm.

$25.2-6$

As in exercise $25.1-10$ a negative-weight cycle can be determined by looking at the diagonal of the output matrix.

$25.2-8$

We wish to compute the transitive closure of a directed graph $G = (V, E)$. Construct a new graph $G^* = (V, E^*)$ where E^* is initially empty. For each vertex v traverse the graph G adding edges for every node encountered in E^*. This takes $O(VE)$ time.

$25-1$

We consider the problem of dynamically maintaining the transitive closure of a graph $G = (V, E)$ represented by a boolean matrix B. First notice that given two connected components C_1 and C_2

a. For a connected component C in G we have $B_{ij} = 1$ if $i, j \in C \times C$. Thus for two connected components C_1 and C_2 with no edges between them we can compute $C_1 \cup C_2$ simply by setting $B_{ij} = 1$ if $i, j \in C_1 \cup C_2 \times C_1 \cup C_2$. In the matrix this can be done by computing the bitwise or of row i and j, denoted by r, in B if edge the edge (i, j) is added. The kth bit in r is 1 if and only if $k \in C_1 \cup C_2$. We set each row in $C_1 \cup C_2$ to r.

b. Let $G = (V, E)$ be a graph with an even number of vertices. Let C_1 and C_2 be two connect components partitioning G such that $|C_1| = |C_2| = |V/2|$. Assume there are no edges between C_1 and C_2. Then half of B contains zeros but adding an edge between C_1 and C_2 will leave B consisting completely of zeros. This takes $\Omega(V^2)$ time.

c. Notice that when adding an edge (i, j) we can check if that alters the transitive closure in $O(V)$ time. If the ith and jth row are the same then i and j are in the same connected component and there is no need to update the matrix. Thus we need only perform the $O(V^2)$ update is the connected components are altered. This happens at most $V + 1$ times. The other $O(V^2)$ use $O(V)$ time at most. The total time is $O(V^3)$ for any sequence of n insertions.

$25 - 2$

We consider an ϵ-dense graph $G = (V, E)$ where $|E| = \Theta(V^{1+\epsilon})$.

a. By exercise 6—2 INSERT and DECREASE-KEY can be done in $O(\log_d n)$ time while EXTRACT-MIN takes $O(d \log_d n)$ time. For $d = n^\alpha$ we obtain running times of $1/\alpha$ and n^α/α.

b. Using dijkstra's algorithm the running time depends on the priority queue as follows.

- $|V|\cdot$INSERT
- $|V|\cdot$EXTRACT-MIN
- $|E|\cdot$DECREASE-KEY

With d-ary heaps and $d = \Theta(V^\epsilon)$ we obtain a running time of

$$\frac{VV^\epsilon + E}{\epsilon} = O(E)$$

for constant ϵ, $0 < \epsilon \leqslant 1$.

c. Compute single source shortest path from each vertex $v \in V$ using the algorithm from subexercise b.

26.1 − 1

Assume $(u, v) \notin E$ and $(v, u) \notin E$ then by capacity constraint $f(u, v) \leqslant 0$ and $f(v, u) \leqslant 0$. By Skew symmetry $f(u, v) = f(v, u) = 0$.

26.1 − 6

Let f_1 and f_2 be flows in a flow network $G = (V, E)$. The sum $f_1 + f_2$ is defined by $(f_1 + f_2)(u, v) = f_1(u, v) + f_2(u, v)$ for all $u, v \in V$. Of the tree flow properties the following are satified by $f_1 + f_2$:

Capacity constraint: May clearly be violated.

Skew symmetry: We have:

$$\begin{aligned}(f_1 + f_2)(u, v) &= f_1(u, v) + f_2(u, v) = -f_1(v, u) - f_2(v, u) \\ &= -(f_1(v, u) + f_2(v, u)) = -(f_1 + f_2)(v, u)\end{aligned}$$

Flow conservation: Let $u \in V - s, t$ be given. Then:

$$\begin{aligned}\sum_{v\in V}(f_1 + f_2)(u, v) &= \sum_{v\in V}(f_1(u, v) + f_2(u, v)) = \sum_{v\in V} f_1(u, v) + \sum_{v\in V} f_2(u, v) \\ &= 0 + 0 = 0\end{aligned}$$

26.2 − 4

Prove that for any vertices u and v and any flow and capacity functions f and c we have: $c_f(u, v) + c_f(v, u) = c(u, v) + c(u, v)$. Obvious since:

$$\begin{aligned}c_f(u, v) + c_f(v, u) &= c(u, v) - f(u, v) + c(v, u) - f(v, u) \\ &= c(u, v) + c(v, u) - f(u, v) + f(v, u) \\ &= c(u, v) + c(v, u)\end{aligned}$$

26.2 − 7

Show that the function f given by:

$$f_p(u, v) = \begin{cases} c_f(p) & \text{if } (u, v) \text{ is on } p \\ -c_f(p) & \text{if } (v, u) \text{ is on } p \\ 0 & \text{otherwise} \end{cases}$$

is a flow in G_f with value $|f_p| = c_f(p) > 0$. We simply check that the three properties are satified:

Capacity constraint: Since $c_f(p)$ is a minimum capacity on the path p and 0 otherwise, no capacities can be exceeded.

Skew symmetry: If u and v are on p the definition clearly satifies this constraint. Otherwise the flow is 0 and the constraint is again satified.

Flow conservation: Since $c_f(p)$ is constant along a path flow conservation must clearly be preserved.

$26.2 - 9$

We wish to compute the edge connectivity of an undirected graph $G = (V, E)$ by running a maximum-flow algorithm on at most $|V|$ flow networks of the same size as G.

Let G_{uv} be the directed version of G. We will consider G_{uv} as a flow network where $s = u$ and $t = v$. We set the capacity of every edge to 1 so that the number of edges crossing any cut will equal the capacity of the cut. Let f_{uv} be a maximum flow of G_{uv}.

The edge connectivity can now be computed by finding $\min_{v \in V - \{u\}} |f_{uv}|$. This is can easily be seen by using the max-flow min-cut theorem.

$26.3 - 3$

We wish to give an upper bound on the length of any augmenting path found in the G' graph. The augmenting path is a simple path in the residual graph G'_f. The key observation is that edges in the residual graph may go from R to L. Hence a path must be of the form:

$$s \to L \to R \to \ldots \to L \to R \to t$$

Crossing between L and R as many times as it can without using a vertex twice. At most $2 + 2\min(|L|, |R|)$ vertices can be in the path and an upper bound on the length is therefore $2\min(|L|, |R|) + 1$.

32.1 − 2

Assume all the characters of P are different. A mismatch with T a position i of P in line 4 of NAIVE-STRING-MATCHER then implies that mean that we can continue our search from position $s + i$ in T. Thus a linear search of T is sufficient.

32.1 − 4

To determine if a pattern P with gap characters exists in T partition P into substrings $P_1, \ldots, P_k$ determined by the gap characters. Search for P_1 and if found continue searching for P_2 and so on. This clearly find a pattern if one exists.

32.3 − 5

We can construct the finite automaton corresponding to a pattern P using the same idea as in exercise 32.1−4. Partition P into substrings $P_1, \ldots, P_k$ determined by the gap characters. Construct finite automatons for each P_i and combine sequentially, i.e., the accepting state of P_i, $1 \geqslant i < k$ is no longer accepting but has a single transition to P_{i+1}.

32.4 − 2

Since $\pi[q] < q$ we trivially have $|\pi^*[q]| \leqslant q$. This bound is tight as illustrated by the string a^q. Here $\pi[q] = q - 1$, $\pi^{(1)}[q] = q - 2$, and so on resulting in $\pi^*[q] = \{q - 1, \ldots, 0\}$.

32.4 − 3

The indices in which P occurs in PT can be determined as the set $M = \{q \mid m \in \pi^*[q] and q \geqslant 2m\}$.

32.4 − 5

We can determine if T is a cyclic rotation of T′ matching T′ against TT.

33.1 − 4

Show how to determine if three point are collinear in a set of n points. For each point p_0 sort the $n-1$ other points according to the polar angle with respect to p. If two points p_1 and p_2 have the same polar angle then p_0, p_1 and p_2 are collinear. This can approach can be implemented in $O(n^2 \lg n)$.

33.2 − 3

Find two cases where the ANY-SEGMENT-INTERSECT go wrong if used to compute all intersections from left to right.

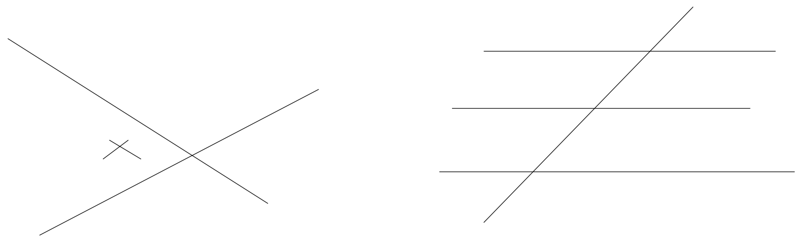

Figure 6: Two cases for the ANY-SEGMENT-INTERSECT

On the first illustration the rightmost intersection is found first and on the other the middle horisontal segment is not found.

33.2 − 4

An n-vertex polygon is simple if and only if no edges intersect. Hence we can simply run ANY-SEGMENT-INTERSECT in $O(n \lg n)$ time to determine if a polygon is simple.

33.2 − 5

To check if two simple n-vertex polygons intersect we can as above simply run ANY-SEGMENT-INTERSECT in $O(n \lg n)$ time.

33.3 − 2

Show that a lower bound of computing a convex hull of n points is $\Omega(n \lg n)$ if the computation model has the same lower bound for sorting.

The n points can be transformed into points in the plane by letting a point i map to the point (i, i^2). Computing the convex hull outputs the points in counterclockwise order and thus sorted.

33 − 1

a. Compute the convex layers of n points in the plane. Let h_i denote the number of points in the ith layer. Initially, compute the convex hull using Jarvis' march. Removing these points and repeating the procedure until no points exists does the job since $\sum h_i = n$ and the complete running time therefore is $O(n^2)$.

b. Clearly, if a lower bound for computing the convex hull is $\Omega(n \lg n)$ by exercise 33.3 − 2 this must also be a lower bound for computing the convex layers.

$34.1-4$

The dynamic programming algorithm for the knapsack problem runs in time $O(nW)$ where n is the number of items and W is the maximun weight of the items. Since W do not depend on the size of the input the algorithm is not polynomial.

$34.1-5$

Consider an algorithm that calls $O(n)$ subroutines each taking linear time. The first call can produce $O(n)$ output which can be concatenated to the original input and used as input to the next giving it time $O(2n)$ and sofort. The total time used is then $\sum_{k=1}^{n} 2^k n$ which is clearly not polynomial. If however we only call a constant number of subroutines the algorithm will be polynomial.

$34.1-6$

Assume that L, L_1, $L_2 \in P$. The following statements hold:

- $L_1 \cup L_2 \in P$ since we can decide if $x \in L_1 \cup L_2$ by deciding if $x \in L_1$ and then if $x \in L_2$. If either holds then $x \in L_1 \cup L_2$ otherwise it is not.
- $L_1 \cap L_2 \in P$ since we can decide if $x \in L_1$ and then if $x \in L_2$. If both holds then $x \in L_1 \cap L_2$ otherwise it is not.
- $\overline{L} \in P$ since $x \in \overline{L} \iff x \notin L$.
- $L_1 L_2 \in P$. Given a string x of length n denote its substring from index i to j by x_{ij}. We can then decide x by deciding $x_{1k} \in L_1$ and $x_{k+1n} \in L2$ for all the n possible values of k.
- $L^* \in P$. We can prove this showing that the result holds for L^k for all k and thus for $\cup_{i=0}^{k} L_k$. We will use induction on k. If $k = 0$ we only consider the empty language and the result is trivial. Assume that $L^k \in P$ and consider $L^{k+1} = LL^k$. The above result on concatenation gives us that $LL^k \in P$.

$34.2-3$

Assume that HAM-CYCLE $\in P$. First notice that for each nodes exactly two incident edges participate in the cycle. We can find a hamilton cycle as follows. Pick a node $v \in V$ and let E_v be the edges incident to v. Compute a pair $e_1, e_2 \in E_v$ such that $G' = (V, (E - E_v) \cup \{e_1, e_2\}$ contains a hamilton cycle. This can be done in polynomial time by trying all possible pairs. Recursively apply the procedure on another node w for the graph G'. This produces in polynomial time a graph $H = (V, C)$ where $C \subseteq E$ is a hamiltonian cycle.

$34.2-5$

Any NP complete language can be decided by an algorithm running in time $2^{O(n^k)}$ for some constant k simply by trying all the possible certificates.

$34.2-9$

We wish to show that $P \subseteq co-NP$. Assume that $L \in P$. Since P is closed under complement we have that $\overline{L} \in P$ and thus $\overline{L} \in NP$ giving us that $L \in co-NP$.

$34.2-10$

Show that $NP \neq co-NP \implies P \neq NP$. By contraposition the statement is the same as $P = NP \implies NP = co-NP$. Since P is closed under complement the statement is obvious.

34.3 − 2

Show that $\leqslant_p$ is a transitive relation. Let $L_1 \leqslant_p L_2$ and $L_2 \leqslant_p L_3$. Then there exists polynomial-time computable reduction functions f_1 and f_2 such that:

- $x \in L_1 \iff f_1(x) \in L_2$
- $x \in L_2 \iff f_2(x) \in L_3$

The function $f_2(f_1(x))$ is polynomial-time computable and satifies that $x \in L_1 \iff f_2(f_1(x)) \in L_3$ thus giving us that $L_1 \leqslant_p L_3$.

34.3 − 3

Show that $L \leqslant_p \overline{L} \iff \overline{L} \leqslant_p L$. If $L \leqslant_p \overline{L}$ and f is the reduction function then we have by definition that $x \in L \iff f(x) \in \overline{L}$ for some x. This means that $x \notin L \iff f(x) \notin \overline{L}$ which is $x \in \overline{L} \iff f(x) \in L$ giving us that $\overline{L} \leqslant_p L$. The converse can be proved similarly.

34.3 − 6

Assume that $L \in P$. We wish to show that L is P-complete unless it is the empty language or $\{0,1\}^*$. Given $L' \in P$ we can reduce it to L simply by using the polynomial-time algorithm for deciding L' to construct the reduction function f. Given x decide if $x \in L'$ and set $f(x)$ such that $f(x) \in L$ if $x \in L$‘ and $f(x) \notin L$ otherwise. Clearly, this is not possible for the two trivial languages mentioned above.

34.3 − 7

Show that $L \in NPC \iff \overline{L} \in co-NPC$. Assume $L \in NPC$. Then $L \in NP$ giving us that $\overline{L} \in co-NP$. Assume further that $L' \leqslant_p L$ for all $L \in NP$. This means that $x \in L' \iff f(x) \in L$ for some polynomial-time reduction function f. Then we have that $x \in \overline{L'} \iff f(x) \in \overline{L}$ which means that $\overline{L'} \leqslant_p \overline{L}$ for all $\overline{L'} \in co-NP$. The converse can be shown similarly.

34.4 − 5

If a formula is given in disjunctive normal form we can simply check if any of the AND'clauses can be satified to determine if the entire formula can be satified.

34.4 − 7

Show that 2-CNF is solvable in polynomial time. Assume w.l.o.g. that each clause contains exactly 2 literals. Following the hint we construct a directed graph $G = (V, E)$ as follows.

- Let $x_0, \ldots, x_n$ be the variables in the formula. There are two vertices v_i and $\overline{v_i}$ for each x_i. The vertex v_i corresponds to x_i and $\overline{v_i}$ corresponds to $\neg x_i$
- For each clause we construct two edges as in the hint. For example, given for $x_i \vee \neg x_j$ we create (v_j, v_i) $(\overline{v_i}, \overline{v_j})$.

We claim that this formula is satifiable if and only if no pair of complimentary literals are in the same strongly connected component of G. If there are paths from u to v and vice versa, then in any truth assignemt the corresponding literals must have the same value since a path is a chain of implications.

Conversely, suppose no pair of complementary literals are in the same strongly connected component. Consider the dag obtained by contracting each strongly connected component to a single vertex. This dag induces a partial order, which we then extend to a total order using topological sort. For each x_i, if the component of v_i precedes the component of $\overline{v_i}$, set $x_i = 0$ else set $x_i = 1$. We claim that this is a valid truth assignment, i. e., that (i) all literals in the same component are assigned the same values and (ii) if a component B is reachable from A then A, B cannot be assigned 1, 0.

We first prove (i). Assume for the contrary that two literals l_1 and l_2 are in the same strongly connected component S but the strongly connected component containing $\neg l_1$ precedes S in the total order and the component containing $\neg l_2$ is preceded by S. Since l_1 and l_2 are in the same component $l_1 \to l_2$ and $l_2 \to l_1$. It also follows that the clauses $(l_1 \vee \neg l_2)$ and $(\neg l_1 \vee l_2)$ can be obtained. Hence, there must be a path from $\neg l_2$ to $\neg l_1$. This contradicts the total order.

We prove (ii). Assume for contradiction that there are two connected components A and B such that B is reachable from A, but our algorithm assigns 1 and 0 to A and B. Let l_a and l_b be a literals in A and B respectively. Note that there must be a path from $\neg l_b$ to $\neg l_a$. Let $\overline{B}$ and $\overline{A}$ be the component of $\neg l_a$ and $\neg l_b$. Clearly, $\overline{B}$ has value 1 and $\overline{A}$ has value 0. In the total order B preceded $\overline{B}$ and $\overline{A}$ preceded A. This implies that there is a cycle in the total order.

34.5 − 1

Show that the subgraph-isomorphism problem is NP-complete. It is straightforward to show that the problem is in NP. To show that it is NP-hard we reduce from the hamiltonian cycle problem. Let G be a graph G with n vertices. Clearly, G has a hamiltonian cycle if and only if the cycle with n vertices C_n is isomorphic to a subgraph of G.

34.5 − 5

Show that the set-partitioning problem is NP-complete. Clearly, using the partion of the set as certificate shows that the problem is in NP. For the NP-hardness we give a reduction from subset-sum. Given an instance $S = \{x_1, \dots, x_n\}$ and t compute r such that $r + t = (\sum_{x \in S} x + r)/2$ $(r = \sum_{x \in S} x - 2t)$. The instance for set-partition is then $R = \{x_1, \dots, x_n, r\}$. We claim that S has a subset summing to t if and only if R can be partitioned. Assume S has a subset S' summing to t. Then the set $S' \cup \{r\}$ sums to $r + t$ and thus partitions R. Conversely, if R can be partitioned then the set containing the element r sums to $r + t$. All other elements in this set sum to t thus providing the subset S'.

34.5 − 6

By exercise 34.2 − 6 the hamiltonian-path problem is in NP. To show that the problem is NP-hard construct a reduction from the hamilton-cycle problem. Given a graph G pick any vertex v and make a “copy” of v, u that is connected to the same vertices as v. It can be shown that this graph has a hamiltonian path from v to u if and only if G has a hamilton-cycle.

Notes for the exercises

- The solution to exercise 34.4 − 7 is largely taken from a solution given by Edith Cohen.

$35.1 - 1$

A graph with two vertices and an edge between them has the optimal vertex cover of consisting of a single vertex. However, APPROX-VERTEX-COVER returns both vertices in this case.

$35.1 - 3$

The following graph will not yield an ratio bound of two using the proposed heuristic. The vertices are $V = \{a1, a2, a3, a4, a5, a6, a7, b1, b2, b3, b4, b5, b6, c1, c2, c3, c4, c5, c6\}$ and the adjancancy list representation is given by:

$a1$: $b1, b2$

$a2$: $b3, b4$

$a3$: $b5, b6$

$a4$: $b1, b2, b3$

$a5$: $b4, b5, b6$

$a6$: $b1, b2, b3, b4$

$a7$: $b2, b3, b4, b5, b6$

Additionally there should be an edge from $b1$ to $c1$ and from $b2$ to $c2$ and so forth.

The heuristic is actually a $\Theta(\log n)$ approximation algorithm. For the upper bound note that the algorithm corresponds to the greedy set cover algorihm, where edges are the elements to be covered and each node v represents a set containing the edges incident to this node. By corollary 35.5 we get the upper bound. For the lower bound consider the bipartite graph $G = (V \cup W, E)$ constructed as follows. Initially, let $V = \{v_1, \ldots v_n\}$ and $W = \{w_1, \ldots w_n\}$. First connect v_i to w_i for all i. Then add $\lfloor n/2 \rfloor$ nodes to W and connect each of these to exactly two nodes of V not connected to any other of the added nodes. Simirlarly, continue adding $\lfloor n/3 \rfloor$ nodes an so on. Finally, V contains n nodes and W contains $\Omega(nH_n)$ nodes. Clearly, V is the optimal vertex cover, however the algorithm will (if unlucky) select W. Thus we have shown a lower bound of $\Omega(\log n)$ on the approximation factor.

$35.1 - 4$

To construct an optimal vertex cover for a tree simply pick a node v such that v has at least one leaf. Select v and remove v and all vertices and edges incident to v and continue until no more vertices are left. Since the edges incident to the leafs of v needs to be covered and we can do this optimally by selecting v the algorithm finds the optimal cover.

$35.1 - 5$

The clique problem and vertex cover problem is related through a reduction. This, however, does not imply that there is a constant ratio bound approximation algorithm since the reductions may alter the ratio polynomially.

$35.2 - 2$

We transform an instance of the travelling salesman into another instance that satifies the triangle equality. Let k be the sum of weights of all edges. By adding k to all edges we obtain an instance that satifies the triangle inequality since k dominates the sums. The optimal tour remains unchanged since all tours contain the same number of edges and we therefore simply added nk to all tours. This does not contradict theorem 35.3 since the ratio bound of the transformed problem does not give a constant ratio bound on the original problem.

$35.5 - 4$

We can modify the approximation scheme to find a value greater than t that is the sum of some subset S' of a list S by running the approximation algorithm and then summing the elements of the complement list $S - S'$.