Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-1.7

The numbers in the first row are quite large. The table below calculates it approximately in powers of 10. People might also choose to use powers of 2. Being close to the answer is enough for the big numbers (within a few factors of 10 from the answers shown).

	1 Second	1 Hour	1 Month	1 Century
$\log n$	$2^{10^6} \approx 10^{300000}$	$2^{3.6\times10^9} \approx 10^{10^9}$	$2^{2.6\times10^{12}} \approx 10^{0.8\times10^{12}}$	$2^{3.1\times10^{15}} \approx 10^{10^{15}}$
$\sqrt{n}$	$\approx 10^{12}$	$\approx 1.3\times10^{19}$	$\approx 6.8\times10^{24}$	$\approx 9.7\times10^{30}$
n	10^6	3.6×10^9	$\approx 2.6\times10^{12}$	$\approx 3.12\times10^{15}$
$n\log n$	$\approx 10^5$	$\approx 10^9$	$\approx 10^11$	$\approx 10^{14}$
n^2	1000	6×10^4	$\approx 1.6\times10^6$	$\approx 5.6\times10^7$
n^3	100	≈ 1500	≈ 14000	≈ 1500000
2^n	19	31	41	51
$n!$	9	12	15	17

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-1.10

The Loop1 method runs in $O(n)$ time.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-1.11

The Loop2 method runs in $O(n)$ time.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-1.12

The Loop3 method runs in $O(n^2)$ time.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-1.13

The Loop4 method runs in $O(n^2)$ time.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-1.19

By the definition of big-Oh, we need to find a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $(n+1)^5 \leq c(n^5)$ for every integer $n \geq n_0$. Since $(n+1)^5 = n^5 + 5n^4 + 10n^3 + 10n^2 + 5n + 1$, $(n+1)^5 \leq c(n^5)$ for $c = 8$ and $n \geq n_0 = 2$.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-1.23

By the definition of big-Omega, we need to find a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $n^3 \log n \geq cn^3$ for $n \geq n_0$. Choosing $c = 1$ and $n_0 = 2$, shows $n^3 \log n \geq cn^3$ for $n \geq n_0$, since $\log n >\geq 1$ in this range.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-1.7

To say that Al's algorithm is "big-oh" of Bill's algorithm implies that Al's algorithm will run faster than Bill's for all input greater than somenon-zero positive integer n_0. In this case, $n_0 = 100$.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-1.8

Since r is represented with 100 bits, any candidate p that the eavedropper might use to try to divide r uses also at most 100 bits. Thus, this very naive algorithm requires 2^{100} divisions, which would take about 2^{80} seconds, or at least 2^{55} years. Even if the eavesdropper uses the fact that a candidate p need not ever be more than 50 bits, the problem is still difficult. For in this case, 2^{50} divisions would take about 2^{30} seconds, or about 34 years.

Since each division takes time $O(n)$ and there are 2^{4n} total divisions, the asymptotic running time is $O(n \cdot 2^{4n})$.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-1.9

One possible solution is $f(n) = n^2 + (1 + sin(n))$.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-1.17

The induction assumes that the set of $n-1$ sheep without a and the set of $n-1$ sheep without b have sheep in common. Clearly this is not true with the case of 2 sheep. If a base case of 2 sheep could be shown, then the induction would be valid.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-1.22

$$\sum_{i=1}^{n} n \log_2 i < \sum_{i=1}^{n} n \log_2 n = n \log_2 n$$

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-1.23

For convenience assume that n is even. Then

$$\sum_{i=1}^{n} \log_2 i/geq \sum_{i=\frac{n}{2}+1}^{n} \log_2 \frac{n}{2} = \frac{n}{2} \log_2 \frac{n}{2},$$

which is $\Omega(n \log n)$.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-1.25

Let B_{n-1}, B_{n-2},..., B_1, B_0 be the n bottles of wine. Consider any binary string $\mathbf{b} = b_{n-1}b_{n-2}...b_1b_0$ of length n. A string $\mathbf{b}$ corresponds to a test as follows. If $\mathbf{b}$ has k 1's, we give a taste tester to drink a sample that consists of exactly k drops: a drop of bottle B_i is included to the sample only if $b_i = 1$. The idea is to have that sufficient collection of tests $\mathbf{b_1}$, $\mathbf{b_2}$,..., $\mathbf{b_T}$ running in parallel, so that, when (after a month) their outcome is known, we can deterministically identify the poisoned bottle.

There are several ways for someone to construct the tests. A naive solution uses n tests, each of them having a drop from only one distinct bottle. We can do much better. An efficient construction of the tests will be in such a way so that a binary search is performed in the sequence of the bottles.

For simplicity assume that n is a power of 2, i.e., $n = 2^k$ for some integer k. We define $\mathbf{B}(t)$ to be the binary string of length t that consists of $\frac{t}{2}$ consecutive 0's and followed by $\frac{t}{2}$ 1's. Let the first test be $\mathbf{b_1} = \mathbf{B}(n)$. Given the output of this test, our search space is reduced by half: if $\mathbf{b_1}$ is positive then we know that the poisoned bottle is one of $B_{\frac{n}{2}-1}$,...,B_1,B_0, otherwise one of B_{n-1},B_{n-2},...,$B_{\frac{n}{2}}$. Let now the second test be $\mathbf{b_2} = \mathbf{B}(\frac{n}{2})||\mathbf{B}(\frac{n}{2})$, where $||$ denotes string concatenation. Clearly tests $\mathbf{b_1}$ and $\mathbf{b_2}$ reduce the search space to one forth of the initial. We proceed in the same way: $\mathbf{b_3} = \mathbf{B}(\frac{n}{4})||\mathbf{B}(\frac{n}{4})||\mathbf{B}(\frac{n}{4})||\mathbf{B}(\frac{n}{4})$ and, generally, $\mathbf{b_i} = \mathbf{B}(\frac{n}{i+1})||\mathbf{B}(\frac{n}{i+1})||\mathbf{B}(\frac{n}{i+1})||\mathbf{B}(\frac{n}{i+1})$, for $1 \leq i \leq k$, where $k = \log n$. Combining all k tests $\mathbf{b_1}$, $\mathbf{b_2}$, ..., $\mathbf{b_k}$ the search space is finally reduced to only one bottle. A similar reasoning can be followed in the case that n is not an exact power of 2; in that case, the number of tests are $\lceil n \rceil$.

We give an example for $n = 13$. The tests could be:

test 1: $\mathbf{b_1}$=11111100 0 0000

test 2: $\mathbf{b_2}$=11100011 1 0000

test 3: $\mathbf{b_3}$=10010010 0 1100

test 4: $\mathbf{b_4}$=11011011 0 1010,

that is, we use 4 taste testers. Suppose that only test 2 is positive. Given the structure of the tests, we can infer that the 5th bottle B_4 is the poisoned one.

When n is not a power of 2, there are generally more that one possible test structures. They all can determine the poisoned bottle, but the king should choose that collection of tests that have the least number of 1's (why?)...

Why is the king not smiling? No, it's not because the rumour about the queen - everybody believes she organized this!... The king is unhappy because all the bottles of wine (some of them were really old) had to be opened.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-1.27

Start at the upper left of the matrix. Walk across the matrix until a 0 is found. Then walk down the matrix until a 1 is found. This is repeated until the last row or column is encountered. The row with the most 1's is the last row which was walked across.

Clearly this is an $O(n)$-time algorithm since at most $2 \cdot n$ comparisons are made.

Algorithm Design
M. T. Goodrich and R. Tamassia
John Wiley & Sons
Solution of Exercise C-1.28

Using the two properties of the array, the method is described as follows.

- Starting from element $A[n-1,0]$, we scan A moving only to the right and upwards.
- If the number at $A[i,j]$ is 1, then we add the number of 1s in that column $(i+1)$ to the current total of 1s
- Otherwise we move up one position until we reach another 1.

An example of the traversal of the array is shown in the figure 0.1.

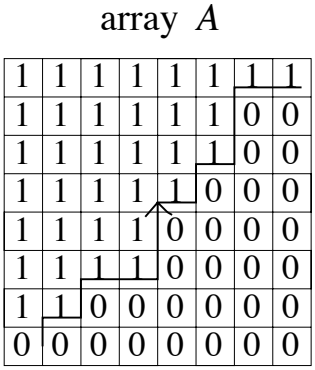


Figure 0.1: The traversal of the array.

The pseudo-code of the algorithm is shown below.

The running time is $O(n)$. In the worst case, you will visit at most $2n-1$ places in the array. In the case that the diagram has all 0s in rows 2 through n and 1s in the first row, then there will be $n-1$ iterations of the **for** loop at constant time (since it will never enter the **while** loop) and 1 iteration of the **while** loop which has n iterations of constant time.

Algorithm NumberOfOnes(A):
Input: An 2D array $n \times n$ A with elements 1s and 0s as described.
Output: The total number of 1s.
$N \leftarrow 0$
$j \leftarrow 0$
for $i \leftarrow n-1$ **to** 0 **do**
  while $j \leq n-1 \wedge A[i,j] = 1$ **do**
    $N \leftarrow N+i+1$
    $j \leftarrow j+1$
return N

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-2.2

Name the two stacks as E and D, for we will enqueue into E and dequeue from D. To implement enqueue(e), simply call E.push(e). To implement dequeue(), simply call D.pop(), provided that D is not empty. If D is empty, iteratively pop every element from E and push it onto D, until E is empty, and then call D.pop(). For the amortized analysis, charge \$2 to each enqueue, using \$1 to do the push into E. Imagine that we store the extra cyber-dollar with the element just pushed. We use the cyber-dollar associated with an element when we move it from E to D. This is sufficient, since we never move elements back from D to E. Finally, we charge \$1 for each dequeue to pay for the push from D (to remove the element returned). The total charges for n operations is $O(n)$; hence, each operation runs in $O(1)$ amortized time.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-2.4

The number of permutations of n numbers is $n! = n(n-1)(n-2)...3 \cdot 2 \cdot 1$. The idea is to compute this product only with increments of a counter. We use a stack for that purpose. Starting with the call Enumerate$(0,S)$, where S is a stack of n elements. the pseudo code should be like that:

```
Algorithm Enumerate(t,S){
  k = S.size();
  while (! S.isEmpty() ) {
    S.pop();
    t++;
    S': a new stack of size k-1
    Enumerate(t,S');
    }
  }
```

Algorithm Design
M. T. Goodrich and R. Tamassia
John Wiley & Sons
Solution of Exercise C-2.5

We use the circular array approach (recall the implementation of a queue, Figure 2.4 of the textbook). For that purpose, we define the two indexes f and l to point to the element of rank 0 and the element of rank $n-1$ respectively, where n is the size of the vector (alternatively, l could point to the array position that is next to element of rank $n-1$). In our algorithms, we use modular N arithmetic (N is the size of the array A used to implement the vector).

We give the pseudo-code for the various methods:

Algorithm size():
  return $N+l-f+1 \bmod N$

Algorithm isEmpty():
return $(l = f-1)$

Algorithm elemAtRank(r):
  if $r < 0 \vee r \geq$ size() **then**
    throw InvalidRankException
  return $A[(f+r) \bmod N]$

Algorithm replaceAtRank(r,e):
  if $r < 0 \vee r \geq \text{size}()$ **then**
    throw InvalidRankException
  $o \leftarrow A[(f+r) \bmod N]$
  $A[(f+r) \bmod N] \leftarrow e$
  return o

Algorithm insertAtRank(r,e):
  $s \leftarrow \text{size}()$
  if $s = N-1$ **then**
    throw VectorFullException
  if $r < 0 \vee r \geq \text{size}()$ **then**
    throw InvalidRankException
  if $r < \lfloor \frac{s}{2} \rfloor$ **then**
    for $i \leftarrow f, (f+1) \bmod N, ..., (f+r-1) \bmod N$ **do**
      $A[(i-1) \bmod N] \leftarrow A[i]$
    $A[(f+r-1) \bmod N] \leftarrow e$
    $f \leftarrow (f-1) \bmod N$
  else
    for $i \leftarrow l, (l-1) \bmod N, ..., (l-s+r+1) \bmod N$ **do**
      $A[(i+1) \bmod N] \leftarrow A[i]$
    $A[(l-s+r+1) \bmod N] \leftarrow e$
    $l \leftarrow (l+1) \bmod N$

Algorithm removeAtRank(r):
  $s \leftarrow \text{size}()$
  if isEmpty() **then**
    throw VectorEmptyException
  if $r < 0 \vee r \geq \text{size}()$ **then**
    throw InvalidRankException
  $o \leftarrow \text{elemAtRank}(r)$
  if $r < \lfloor \frac{s}{2} \rfloor$ **then**
    for $i \leftarrow (f+r-1) \bmod N, (f+r) \bmod N, ..., (f+1) \bmod N, f$ **do**
      $A[(i+1) \bmod N] \leftarrow A[i]$
    $f \leftarrow (f+1) \bmod N$
  else
    for $i \leftarrow (l-s+1) \bmod N, (l-s) \bmod N, ..., (l-1) \bmod N, l$ **do**
      $A[(i-1) \bmod N] \leftarrow A[i]$

$l \leftarrow (l-1) \bmod N$
return o

Algorithm Design
M. T. Goodrich and R. Tamassia
John Wiley & Sons
Solution of Exercise C-2.9

Algorithm preorderNext(Node v):
    if *v*isInternal() **then**
        return *v*'s left child
    else
        Node p = parent of v
        if v is left child of p **then**
            return right child of p
        else
            while v is **not** left child of p **do**
                $v = p$
                $p = p.parent$
            return right child of p

Algorithm inorderNext(Node v):
    if *v*isInternal() **then**
        return *v*'s right child
    else
        Node p = parent of v
        if v is left child of p **then**
            return p
        else
            while v is **not** left child of p **do**
                $v = p$
                $p = p.parent$
            return p

The worst case running times for these algorithms are all $O(\log n)$ where n is the height of the tree T.

```
Algorithm postorderNext(Node v):
    if visInternal() then
        p = parent of v
        if v = right child of p then
            return p
        else
            v = right child of p
            while v is not external do
                v = leftchildofv
            return v
    else
        p = parent of v
        if v is left child of p then
            return right child of p
        else
            return p
```

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-2.10

The idea is to perform a preorder (or postorder) traversal of the tree, where the "visit" action is to report the depth of the node that is currently visited. This can be easily done by using a counter that keeps track of the current depth. Method PrintDepth(v,d) prints the depth d of the current node v and recursively traverses the subtree defined by v. If T is the tree, then, initially, we call PrintDepth(T.root(),0). Observe that we use only methods of the abstract tree ADT.

Algorithm PrintDepth(v,d):
  print(d)
  for each $w \in T$.children(v) **do**
    PrintDepth(w,$d+1$)

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-2.11

One way to do this is the following: in the ***external*** method, set height and balance to be zero. Then, alter the ***right*** method as follows:

Algorithm right():
  if $v$isInternal(v) **then**
    if $v.leftchild.height > v.rightchild.height$ **then**
      $v.height = v.leftchild.height + 1$;
    else
      $v.height = v.rightchild.height + 1$;
    $v.balance = absval(v.rightchild.height - v.leftchild.height)$;
  printBalanceFactor(v)

Algorithm Design
M. T. Goodrich and R. Tamassia
John Wiley & Sons
Solution of Exercise C-2.13

Examining the Euler tree traversal, we have the following method for finding $\mathsf{tourNext}(v,\alpha)$

- If v is a leaf, then:
  - if α = **left**, then w is v and β = **below**,
  - if α = **below**, then w is v and β = **right**,
  - if α = **right**, then
    - if v is a left child, then w is v's parent and β = **below**,
    - if v is a right child, then w is v's parent and β = **right**.
- If v is internal, then:
  - if α = **left**, then β = **left** and w is v's left child,
  - if α = **below**, then β = **left** and w is v's right child,
  - if α = **right**, then
    - if v is a left child, then β = **below** and w is v's parent,
    - if v is a right child, then β = **right** and w is v's parent.

For every node v but the root, we can find whether v is a left or right child, by asking "v=T.leftChild(T.parent(v))". The complexity is always $O(1)$.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-2.16

```
Algorithm eulerTour(Tree T, Position v):
    state ← start
    while state ≠ done do
      if state = start then
        if T.isExternal(v) then
          left action
          below action
          right action
          state ←= done
        else
          left action
          state ← on_the_left
          v ← v.leftchild
      if state = on_the_left then
        if T.isExternal(v) then
          left action
          below action
          right action
          state = from_the_left
          v ← v.parent
        else
          left action
          v ← v.leftchild
      if state = from_the_left then
        below action
        state ← on_the_right
        v ← v.right
      if state = on_the_right then
        if T.isExternal(v) then
          state = from_the_right
          left action
          below action
          right action
          v ← v.parent
        else
          left action
          state ← on_the_left
          v ← v.left
      if state = from_the_right then
        right action
        if T.isRoot(v) then
          state ← done
        else
          if v is left child of parent then
            state ← from_the_left
          else
```

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-2.17

```
Algorithm inorder(Tree T):
    Stack S ← new Stack()
    Node v ← T.root()
    push v
    while S is not empty do
      while v is internal do
        v ← v.left
        push v
      while S is not empty do
        pop v
        visit v
        if v is internal then
          v ← v.right
          push v
        while v is internal do
          v ← v.left
          push v
```

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-2.18

Algorithm levelOrderTraversal(BinaryTree T):
    Queue Q = new Queue()
    Q.enqueue(T.root())
    while Q is not empty **do**
        Node $v \leftarrow Q$.dequeue()
        if T.isInternal(v) **then**
            Q.enqueue($v.leftchild$)
            Q.enqueue($v.rightchild$)

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-2.21

Algorithm LCA(Node v, Node w):

```
int v_depth ← v.depth
int w_depth ← w.depth
while v_depth > w_depth do
    v ← v.parent
while w_depth > v_depth do
    w ← w.parent
while v ≠ w do
    v ← v.parent
    w ← w.parent
return v
```

Algorithm Design
M. T. Goodrich and R. Tamassia
John Wiley & Sons
Solution of Exercise C-2.22

Let d_{uv} be the diameter of T. First observe that both u and v are tree leaves; if not, we can find a path of higher length, only by considering one of u's or v's children. Moreover, one of u, v has to be a leaf of highest depth. This can be proved by contradiction. Consider a tree and some leaf-to-leaf path P that does not include a leaf of highest depth. Then consider a leaf of greatest depth v; it is always possible to find a new path P' that starts at v and is longer than P. This yields the desired contradiction.

The main idea of the algorithm is to find a leaf v of highest depth and starting from this leaf to keep moving towards the tree's root. At each visited node u (including v, but excluding the tree's root), the height of u's sibling is computed and the current length L_{max} of the longest path in which v belongs is updated accordingly. When we are at T's root, L_{max} contains the diameter of T.

We give the pseudo-code for the above algorithm. We use methods depth(T,v) and height(T,v), which are described in section 2.3.2, pages 80-81, of the textbook, and compute the depth and correspondingly the height of the subtree rooted at v. Method DeepestLeaf(T, v, H, c) returns the deepest node of T or NULL if it is not found in the subtree rooted at v. This is done by essentially performing a pre-order traversal that stops as soon as the deepest leaf is found. H is T's height and c is a counter.

```
Algorithm DeepestLeaf(T,v,H,d):
  Input : a tree T, a node v, the height H, and the current depth d
  Output : The deepest node, or NULL if it is not found

  if T.isExternal(v) then
    if d = H then { Base case: check for deepest node }
      return v
    return NULL
  u ← NULL
  {See if there is a deepest leaf in the left child}
  u ← DeepestLeaf(T,T.leftChild(v),H,d+1)
```

```
  if u ≠ NULL then
    return u
  {See if there is a deepest leaf in the right child}
  u ← DeepestLeaf(T,T.rightChild(v),H,d+1)
  return u


Algorithm findDiameter(T):
  Input : a tree T
  Output : The diameter of T

  H ← depth(T,T.root())
  v ← DeepestLeaf(T,v,H,0)
  if v = T.root() then
    return 0 { if deeps leaf is just the root node, diameter is 0}
  L_max ← 0
  x ← 1   {x: number of nodes visited}
  while v ≠ T.root() do
    {Assign y to be the length of the path going to the sibling}
    y ← height(T,T.sibling(v))
    {See if we have found a new longest path}
    if x+1+y > L_max then
      L_max ← x+1+y
    {Climb up the tree one step}
    x ← x+1
    v ← T.parent(v)
  return L_max
```

The running time of the algorithm is linear. Both depth(T,v) and height(T,v) have complexity $O(S_v)$, where S_v is the size of the subtree rooted at v. depth(T,v) is called from the tree's root once, so its time complexity is $O(n)$. height(T,v) is called for each sibling on our way up to the root, so, in total, it adds an $O(n)$ time complexity. DeepestLeaf also has linear worst case time complexity, for it is essentially a pre-order tree traversal. Finally, the leaf-to-root path traversal adds a linear time complexity.

We can improve on this algorithm, however. Note that the above algorithm can visit all the nodes in the tree twice, once to find the deepest node, and once to

find the height subtrees. We can combine these two operations into one algorithm with a little bit of ingenuity and recursion. We observe that given a binary tree T_v rooted at a node v, if we know the diameters and heights of the two subtrees, we know the diameter of T_v. Imagine we took a path from the deepest node of the left subtree to the deepest node of the other subtree, passing through v. This path would have length $= 2 + \mathsf{height}(T_v, T_v.\mathsf{leftChild}(v)) + \mathsf{height}(T_v, T_v.\mathsf{rightChild}(v))$. Further, note that this is a longest path that runs through the root of T_v, since the height of the left and right subtrees is simply the longest path from their respective roots to any of their leafs. Therefore, the longest path in T_v is simply the maximum of longest path in the found in T_v's left and right subtrees, and the longest path running through v. These observations give rise to the following algorithm. For convenience, we denote the return type of this algorithm to be in the form (h,d) where h and d represent height and diameter respectively, and are accessed using the h() and d() methods.

Algorithm Diameter(T,v)
  Input : a tree T and a node in that tree v
  Output : a (h,d) pair

  if T.isExternal(v) **then** {Base case}
    return $(0,0)$

  {Get the return pairs of the left and right subtrees}
  $L \leftarrow$ Diameter(T, T.leftChild(v))
  $R \leftarrow$ Diameter(T, T.rightChild(v))

  {Find the height of this subtree}
  $H \leftarrow \max(L.h(), R.h()) + 1$

  {Find the longest path length}
  $P \leftarrow L.h() + R.h() + 2$
  return $(H, max(L.d(), R.d(), P))$

Since this algorithm visits each node in the tree exactly once, and performs a constant amount of operations at each node, it follows that this algorithm runs in $O(n)$

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-2.24

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-2.29

The path to the last node in the heap is given by the path represented by the binary expansion of n with the highest-order bit removed.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-2.31

Construct a heap, which takes $O(n)$ time. Then call removeMinElement k times, which takes $O(k \log n)$ time.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-2.34

When we remove an item from a hash table that uses linear probing without using deleted element markers, we need to rearrange the hash table contents so that it seems the removed item never existed. This action can be done by simple incrementally stepping forward through the table (much as we do when there are collisions) and moving back one spot each item k for which $f(k)$ equals $f(removeditem)$. We continue walking through the table until we encounter the first item for which the $f(k)$ value differs.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-2.2

The tree expresses the formula $6/(1-5/7)$.

Algorithm Design
M. T. Goodrich and R. Tamassia
John Wiley & Sons
Solution of Exercise R-2.7

We assume that a vector S is used for the representation of the tree T. Also, we assume that S contains as elements positions. We give the pseudo-code for the various methods:

Algorithm root():
  return S.elemAtRank(1)

Algorithm parent(v):
  return S.elemAtRank($\lfloor p(v)/2 \rfloor$)

Algorithm leftchild(v):
  return S.elemAtRank($2p(v)$)

Algorithm rightchild(v):
  return S.elemAtRank($2p(v)+1$)

Algorithm sibling(v):
  if T.isRoot(v) **then**
    throw InvalidNodeException
  if $p(v)$ is even **then**
    return S.elemAtRank($p(v)+1$)
  else
    return S.elemAtRank($p(v)-1$)

Algorithm isInternal(v):
  return $(2p(v)+1 < S.\text{size}()-1) \wedge (S.\text{elemAtRank}(2p(v)) \neq \text{NULL})$

Algorithm isExternal(v):
  return $\neg$ T.isInternal(v)

Algorithm isRoot(v):
  return (v=S.elemAtRank(1))

```
Algorithm size():
  c ← 0
  for i ← 1,2,...,S.size()−1 do
    if S.elemAtRank(i)≠ NULL then
      c ← c+1
  return c
```

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-2.8

221536441039132925
315364410229132925
393644102215132925
391044362215132925
391013362215442925
391013152236442925
391013152236442925
391013152225442936
391013152225294436
391013152225293644

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-2.9

221536441039132925

152236441039132925

152236441039132925

101522364439132925

310152236449132925

391015223644132925

391013152236442925

391013152229364425

391013152225293644

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-2.10

A worst-case sequence for insertion sort would be one that is in descending order of keys, e.g., 443629252215131093. With this sequence, each element will first be moved to the front and then moved back in the sequence incrementally, as every remaining is processed. Thus, each element will be moved n times. For n elements, this means at a total of n^2 times, which implies $\Omega(n^2)$ time overall.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-2.11

The largest key in a heap may be stored at any external node.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-2.13

Yes, the tree T is a min-heap.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-2.14

With a preorder traversal, a heap that produces its elements in sorted order is that which is represented by the vector $(1,2,5,3,4,6,7)$. There does not exist a heap for which an inorder traversal produces the keys in sorted order. This is because in a heap the parent is always less than all of its children or greater than all of its children. The heap represented by $(7,3,6,1,2,4,5)$ is an example of one which produces its keys in sorted order during a postorder traversal.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-2.20

11	39	20	5	16	44	88	12	23	13	94

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-2.21

	20	16	11	39	44	88	12	23	13	94

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-2.22

11	23	20	16	39	44	94	12	88	13	5

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-3.2

There are several solutions. One is to draw the binary search tree created by the input sequence: $9, 5, 12, 7, 13$. Now draw the tree created when you switch the 5 and the 7 in the input sequence: $9, 7, 12, 5, 13$.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-3.3

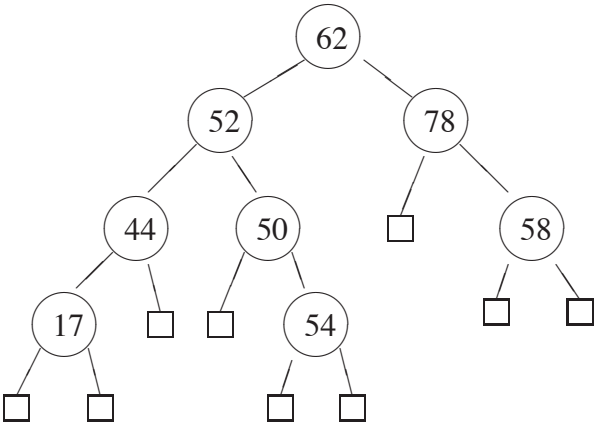
There are several solutions. One is to draw the AVL tree created by the input sequence: $9, 5, 12, 7, 13$. Now draw the tree created when you switch the 5 and the 7 in the input sequence: $9, 7, 12, 5, 13$.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-3.5

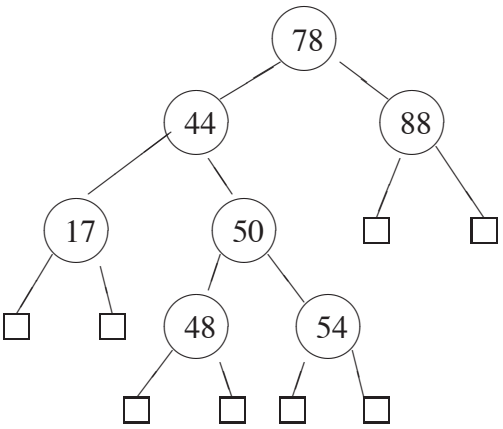


Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-3.6



Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-3.8

No. One property of a $(2,4)$ tree is that all external nodes are at the same depth. The multi-way search tree of the example does not adhere to this property.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-3.9

The key k_2 would be stored at v's parent in this case. This is done in order to maintain the ordering of the keys within the tree. By storing k_2 at v's parent, all of the children would still be ordered such that the children to the left of the key are less than the key and those to the right of the key are greater than the key.

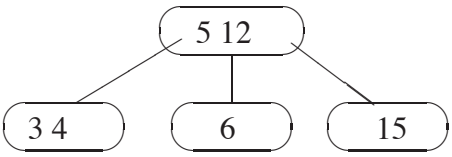
Algorithm Design

M. T. Goodrich and R. Tamassia

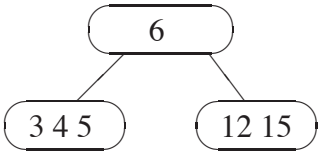
John Wiley & Sons

Solution of Exercise R-3.10

insertion order: 4, 6, 12, 15, 3, 5



insertion order: 12, 3, 6, 4, 5, 15



Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-3.3

Algorithm findAllElements(k, v, c):

Input: The search key k, a node of the binary search tree v and a container c

Output: An iterator containing the found elements

```
if v is an external node then
    return c.elements()
if k = key(v) then
    c.addElement(v)
    return findAllElements(k, T.rightChild(v), c)
else if k < key(v) then
    return findAllElements(k, T.leftChild(v))
else
    {we know k > key(v)}
    return findAllElements(k, T.rightChild(v))
```

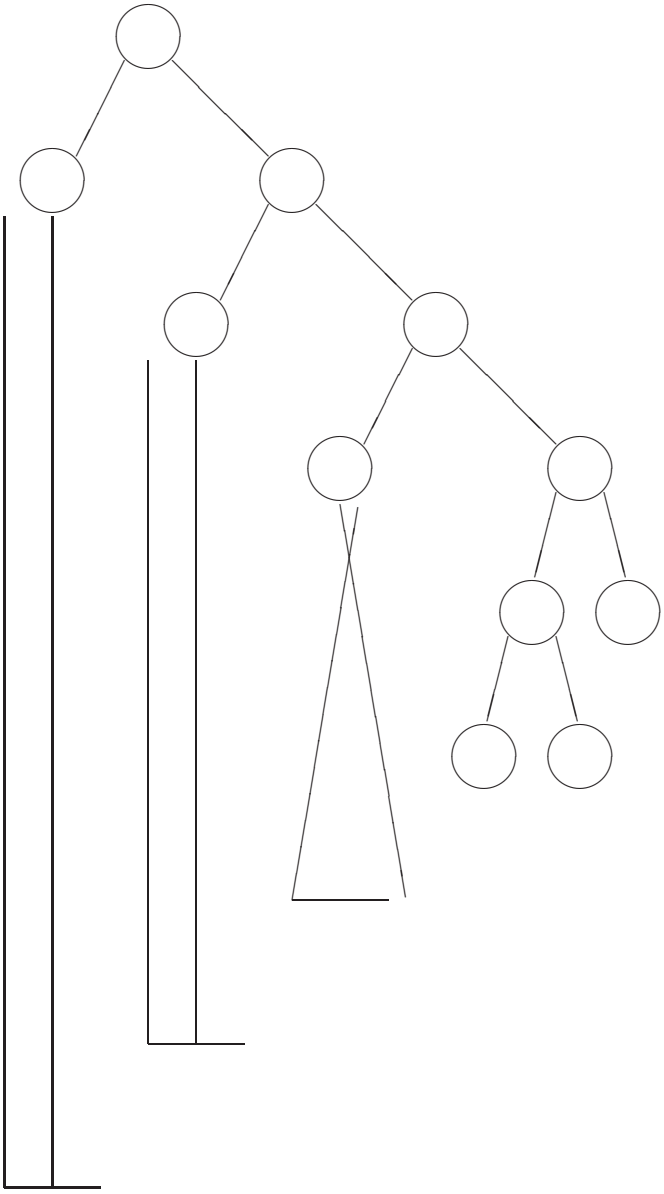
Note that after finding k, if it occurs again, it will be in the left most internal node of the right subtree.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-3.5



Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-3.11

For each node of the tree, maintain the size of the corresponding subtree, defined as the number of internal nodes in that subtree. While performing the search operation in both the insertion and deletion, the subtree sizes can be either incremented or decremented. During the rebalancing, care must be taken to update the subtree sizes of the three nodes involved (labeled a, b, and c by the restructure algorithm).

To calculate the number of nodes in a range (k_1, k_2), search for both k_1 and k_2, and let P_1 and P_2 be the associated search paths. Call v the last node common to the two paths. Traverse path P_1 from v to k_1. For each internal node $w \neq v$ encountered, if the right child of w is in not in P_1, add one plus the size of the subtree of the child to the current sum. Similarly, traverse path P_2 from v to k_2. For each internal node $w \neq v$ encountered, if the left child of w is in not in P_2, add one plus the size of the subtree of the left to the current sum. Finally, add one to the current sum (for the key stored at node v).

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-3.14

Assume, without loss of generality, that $n \geq m$.

Clearly, we can not use the general insertion/deletion tree operations to perform the joining; such a solution would take $O(n \log n)$ time.

Let T and U have heights h_t and h_u respectively. Our task is to "manually" join the two trees into one (2-4) tree V in logarithmic time. V must have all the properties of a (2-4) tree (that is, the size property, the depth property and the property of V being a multi-way search tree). The idea here is very simple. Remove either the largest element of T or the minimum element of U (we name this element e). Without loss of generality, assume that after this removal $h_t \geq h_u$. If $h_t = h_u$, create a new node v that stores the element e that was initially removed and make T's and U's root v's left and right respectively children. If $h_t > h_u$, insert the element e into the rightmost node of tree T at height $h_t - h_u - 1$ and link this node to the root of tree U. If $h_t < h_u$, the approach is symmetric to the one described above.

Readily, the time complexity is $O(\log n)$. Only one element is removed and re-inserted; it can be located in time proportional to tree's height and the insertion and deletion operations take each $O(\log n)$ time. The height of a tree - if it is not stored separately - can be computed in $O(\log n)$ time.

Below, we give pseudo-code for method `join`(T,U). We assume that trees' heights are known. `MostLeftRight`$(T, T$`.root()`$, h, flag)$ returns the rightmost (leftmost) node of tree T at height h, depending on if flag is set (not set).

Algorithm `join`(T,U)**:**
  {remove the largest element from T}
  $h_T \leftarrow$ height of T
  $MaxNodeT \leftarrow$ `MostLeftRight`$(T, T$`.root()`$, h_T, true)$
  $MaxKeyT \leftarrow T$`.Key`$(MaxNodeT,$`Type`$(MaxNodeT))$
  $MaxElemT \leftarrow T$`.Elem`$(MaxNodeT,$`Type`$(MaxNodeT))$
  T`.Remove`$(MaxNodeT, MaxKey, MaxElemT)$
  T`.Restructure`$(MaxNodeT)$

```
{we consider three cases}
h_T ← height of T
h_U ← height of U

if h_T = h_U then {case 1}
V ← new tree
V.Insert(V.root(),MaxKeyT,MaxElemT)
V.Child(V.root(),1) ← T.root()
V.Child(V.root(),2) ← U.root()
return V
if h_T > h_U then {case 2}
v ← MostLeftRight(T,T.root(),h_T − h_U − 1,false)
T.Insert(v,MaxKeyT,MaxElemT)
T.Child(v,Type(v)+1) ← U.root()
T.Restructure(v)
return T
if h_T < h_U then {case 2}
v ← MostLeftRight(U,U.root(),h_U − h_T − 1,true)
U.Insert(v,MaxKeyT,MaxElemT)
U.Child(v,1) ← T.root()
U.Restructure(v)
return U
```

Algorithm MostLeftRight$(T,v,h,flag)$**:**

```
if h = 0 then
  return v
if flag then
  return MostLeftRight(T,T.Child(v,T.Type(v)),h − 1,flag)
else
  return MostLeftRight(T,T.Child(v,1),h − 1,flag)
```

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-3.17

We give two possible solutions. Let v be a red node of a red-black tree T.

- This solution assumes that keys in T are integers. We change keys stored in T using the following simple rule: if v is red, set $\mathsf{new\_key}(v) = 2\mathsf{key}(v)+1$, or if v is black, set $\mathsf{new\_key}(v) = 2\mathsf{key}(v)$. Observe that this "mapping" function is 1-1 and strictly increasing; this property preserves the property of T of being a binary search tree and helps in uniquely identify a node's color. Namely, any node can be identified of being red or black, by simply checking whether $\mathsf{new\_key}(v)$ is respectively odd or even.
- This solution assumes that keys in T are unique. Because of the fact that v is red, it can not be a leaf. Let u and w be v's left and right child respectively. If both u and w are not place-holder nodes, we can represent the fact that v is red, implicitly, by exchanging the children of v, i.e., by making u v's right child and w v's left child. This allows us to correctly identify if a node is red or black, only by checking the key values for nodes u, v and w. Node v is red, if and only if the property of T of being binary search is locally violated, i.e., if and only if $\mathsf{key}(u) > \mathsf{key}(v) > \mathsf{key}(w)$. If one of v's children is a place holder node, then again a similar violation of T's property of being a binary tree can be used to identify v's color. Finally, if both children of v are place-holder nodes, then we can either store some fictitious element in one of them or remove v's right (or left) child.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-3.26

First, observe that methods SkipSearch(k) uses only methods below(p) and after(p). It is a top-down, scan-forward traversal technique. We examine how an insertion and a removal of an item can be performed in a similar traversing manner.

Insertion: The main idea here is that we can do all the coin flips before we do any skip-list traversals or insertions. The insertion algorithm consists of the following steps:

- Flip the coin several times until heads come up to compute the height of the new tower.
- Compute the height of the skip-list if necessary (unless you store this information and update it during insertion/removal stage). We can always compute the height by going down from the topmost left element till we hit the rock bottom.
- If the number of coin tosses is more than the height of the skip-list, add new level(s), otherwise find the topmost level in which to start insertion (i.e., go down height-of-the-list - number-coin-tosses levels).
- Starting from the position found in the previous step, do something very similar to SkipSearch except that you also have to insert the new element at each level before you skip down.

Removal: For the removal algorithm, we perform a SkipSearch(k)-like traversal down the skip-list for the topmost leftmost item such that the item to the right has key k (if such a key exists). We go on by deleting the whole tower in a top-down fashion.

In both cases, we need to restructure the references between elements (namely references below(p) and after(p)). We give the pseudo-code that performs these operations. We assume topleft stores the topmost-left position of the skip-list.

Algorithm insertItem(k, e)

```
{compute the height of the tower}
ctosses ← 0
while (random() < 1/2) do
  ctosses ← ctosses + 1
height ← −1
current ← topleft

{find height of skip list}
while (below(current) ≠ null) do
  current ← below(current)
  height ← height + 1

{ Insert new levels if necessary and find the first level to insert at}
if (height < ctosses) then
  for i ← 1 to ctosses − height do
    oldtopleft ← topleft
    topleft ← new Item(−∞,null)
    after(topleft) ← new Item(∞,null)
    below(topleft) ← oldtopleft
    below(after(topleft)) ← after(oldtopleft)
  insertat ← below(topleft)
else
  insertat ← topleft
  for i ← 0 to (height − ctosses − 1) do
        insertat ← below(insertat)

{ Now do SkipSearch, inserting before going down }
oldnewnode ← null
while (insertat ≠ null) do
  while(key(after(insertat)) ≤ k) do
        insertat ← after(insertat)
  newnode ← new Item(k,e)
  after(newnode) ← after(insertat)
  if (oldnewnode ≠ null) then
        below(oldnewnode) ← newnode
  after(insertat) ← newnode
  oldnewnode ← newnode
  insertat ← below(insertat)
```

```
Algorithm removeElement(k)
  current ← topleft
  while (below(current) ≠ null) do
      current ← below(current)
      while key(after(current)) < k do
              current ← after(current)
      if after(current) = k then
              tmp ← after(current)
              after(current) ← after(after(current))
              delete tmp
```

Note that the insertion has $O(\log(n))$ expected time (for the same reasons that SkipSearch has $O(\log(n))$ expected time.

An alternative realization of insertItem could involve the use of a stack. As we perform SkipSearch, we push into a stack every element that we access. Once we are at the bottom of the structure, we start tossing the coin and as long as the flip is coming up "heads" we insert a copy of the inserted element. We track any after(p) reference that needs be updated by performing a pop operation of the stack. Any below(p) reference is handled by storing the last copy added to the tower. Depending on when the flip is coming up "tails", the stack may not become empty or we may need add new levels. The details are left as an exercise.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-3.30

To implement an efficient ordered dictionary, we can use a hash table with separate chaining. By using this data structure, we will be able to group together all elements with the same key—they will all be chained together. The chains will also be sorted, so as to help with the ordered structure. In this system, all ordered dictionary operations can be handled in $O(\log k + s)$ expected time - a quick hash to the right spot in the table (constant time) plus a quick search through the chain to find the correct element (which takes $O(\log k)$ time).

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-4.1

For each element in a sequence of size n there is exactly one exterior node in the merge-sort tree associated with it. Since the merge-sort tree is binary with exactly n exterior nodes, we know it has height $\lceil \log n \rceil$.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-4.5

Merge sequences A and B into a new sequence C (i.e., call merge(A,B,C)). Do a linear scan through the sequence C removing all duplicate elements (i.e., if the next element is equal to the current element, remove it).

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-4.9

$O(n \log n)$ time.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-4.14

The bubble-sort and merge-sort algorithms are stable.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-4.16

No. Bucket-sort does not use a constant amount of additional storage. It uses $O(n+N)$ space.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-4.1

This can be done by extending the Merger class to be an equalsMerger class. In the equalsMerger class, the methods firstIsLess and firstIsGreater would be null methods. The method bothAreEqual would add a to the Sequence C. Then, after this is done, C could be checked against A and B to see that it is the same size.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-4.4

First we sort the objects of A. Then we can walk through the sorted sequence and remove all duplicates. This takes $O(n \log n)$ time to sort and n time to remove the duplicates. Overall, therefore, this is an $O(n \log n)$-time method.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-4.9

For the red and blue elements, we can order them by doing the following. Start with a marker at the begining of the array and one at the end of the array. While the first marker is at a blue element, continue incrementing its index. Likewise, when the second marker is at a red element, continue decrementing its index. When the first marker has reached a red element and the second a blue element, swap the elements. Continue moving the markers and swapping until they meet. At this point, the sequence is ordered. With three colors in the sequence, we can order it by doing the above algorithm twice. In the first run, we will move one color to the front, swapping back elements of the other two colors. Then we can start at the end of the first run and swap the elements of the other two colors in exactly the same way as before. Only this time the first marker will begin where it stopped at the end of the first run.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-4.10

First sort the sequence S by the candidate's ID. Then walk through the sorted sequence, storing the current max count and the count of the current candidate ID as you go. When you move on to a new ID, check it against the current max and replace the max if necessary.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-4.11

In this case we can store candidate ID's in a balanced search tree, such as an AVL tree or red-black tree, where in addition to each ID we store in this tree the number of votes that ID has received. Initially, all such counts are 0. Then, we traverse the sequence of votes, incrementing the count for the appropriate ID with each vote. Since this data structure stored k elements, each such search and update takes $O(\log k)$ time. Thus, the total time is $O(n \log k)$.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-4.14

To sort S, do a radix sort on the bits of each of the n elements, viewing them as pairs (i, j) such that i and j are integers in the range $[0, n-1]$.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-4.16

Sort the elements of S, which takes $O(n \log n)$ time. Then, step through the sequence looking for two consecutive elements that are equal, which takes an additional $O(n)$ time. Overall, this method takes $O(n \log n)$ time.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-4.21

Any reverse-ordered sequence (in descending order) will fit this solution.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-5.8

Employ the trick used in Java, C, and C++, where we terminate an OR early if one its terms is 1. The probability that one of the terms in each row-column product is $1/k^2$; hence, the expected time for any row-column product is constant. Therefore, the expected running time of the entire product of A and B is $O(n^2)$. If k is n, on the other hand, then the probability than an entry in a row-column product is 1 is $1/n^2$; hence, the expected running time for a row-column product is $O(n)$ in this case. That is, in this case, the expected time to multiply A and B is $O(n^3)$.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-5.12

This is a knapsack problem, where the weight of the sack is n, and each bid i corresponds to an item of weight k_i and value d_i. If each bidder i is unwilling to accept fewer than k_i widgets, then this is a 0/1 problem. If bidders are willing to accept partial lots, on the other hand, then this is a fractional version of the knapsack problem.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-5.3

First give as many quarters as possible, then dimes, then nickles and finally pennies.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-5.4

If the demoninations are \$0.25, \$0.24, \$0.01, then a greedy algorithm for making change for 48 cents would give 1 quarter and 23 pennies.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-5.5

We can use a greedy algorithm, which seeks to cover all the designated points on L with the fewest number of length-2 intervals (for such an interval is the distance one guard can protect). This greedy algorithm starts with x_0 and covers all the points that are within distance 2 of x_0. If x_i is the next uncovered point, then we repeat this same covering step starting from x_i. We then repeat this process until we have covered all the points in X.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-5.6

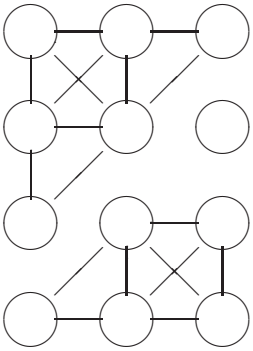
Divide the set of numbers into $n/2$ groups of two elements each. With $n/2$ comparisons we will determine a minimum and a maximum in each group. Separate the maximums and minimums, and find the minimum of the minimums and the maximum of the maximums using the standard algorithm. These additional scans use $n/2$ comparisons each.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-6.1



If G has 12 vertices and 3 connected components, then the maximum number of edges it can have is 45. This would be a fully connected component with 10 vertices (thus having 45 edges) and two connected components each with a single vertex.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-6.2

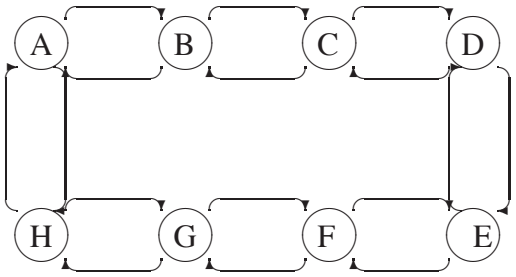
We know that $m \leq n \cdot (n-1)/2$ is $O(n^2)$. It follows, therefore, that $O(\log(n^2)) = O(2\log n) = O(\log n)$.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-6.3



The Euler tour is $A \to B \to C \to D \to E \to F \to G \to H \to A \to H \to G \to F \to E \to D \to C \to B \to A$

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-6.5

Inserting a vertex runs in $O(1)$ time since it is simply inserting an element into a doubly linked list. To remove a vertex, on the other hand, we must inspect all edges. Clearly this will take $O(m)$ time.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-6.7

1. The adjacency list structure is preferable. Indeed, the adjacency matrix structure wastes a lot of space. It allocates entries for 100,000,000 edges while the graph has only 20,000 edges.
2. In general, both structures work well in this case. Regarding the space requirement, there is no clear winner. Note that the exact space usage of the two structures depends on the implementation details. The adjacency matrix structure is much better for operation areAdjacent, while the adjacency list structure is much better for operations insertVertex and removeVertex.
3. The adjacency matrix structure is preferable. Indeed, it supports operation areAdjacent in $O(1)$ time, irrespectively of the number of vertices or edges.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise R-6.10

(BOS, JFK, MIA, ORD, DFW, SFO, LAX).

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-6.12

We can compute the *diameter* of the tree T by slightly modifying the algorithm we used in previous question.

1. Remove all leaves of T. Let the remaining tree be T_1.
2. Remove all leaves of T_1. Let the remaining tree be T_2.
3. Repeat the "remove" operation as follows: Remove all leaves of T_i. Let remaining tree be T_{i+1}.
4. When the remaining tree has only one node or two nodes, stop! Suppose now the remaining tree is T_k.
5. If T_k has only one node, that is the center of T. The *diameter* of T is $2k$.
6. If T_k has two nodes, either can be the center of T. The *diameter* of T is $2k+1$.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-6.13

For each vertex v, perform a modified BFS traversal starting a v that stops as soon as level 4 is computed.

Alternatively, for each vertex v, call method DFS$(v,4)$ given below, which is a modified DFS traversal starting at v:

Algorithm DFS(v,i):
  if $i > 0$ and v is not marked **then**
    Mark v.
    Print v.
    for all vertices w adjacent to v **do**
      DFS$(w, i-1)$.

Algorithm Design
M. T. Goodrich and R. Tamassia
John Wiley & Sons
Solution of Exercise C-6.17

Essentially, we are performing a DFS-like traversal of the graph. The idea is to keep visiting nodes using edges that have never been traversed (in this direction) in such a way so that we are able to finish the traversal exactly at the node where we started from. To accomplish this, we keep track of the edge that we used to reach a node for the very first time. We use this edge to leave the node when we have traversed all its other edges.

The algorithm can be described as follows:

> Start from any vertex, say s, of the graph. Traverse any incident edge of s and visit a new node. On a node that you have just reached, if it has not ever been visited before, label it as VISITED. Mark the "entrance" edge of any node that is labeled as VISITED (in fact, remember the "entrance" neighbor). While being on a node u, traverse any incident edge other than the "entrance" one, that you have not ever traversed from node u, to visit a neighboring node. If, while being at a node u, you can not traverse any edge (except the "entrance" edge) that you have not traversed before, return to the "entry" neighbor; if $u = s$, there is no "entry" neighbor, so terminate.

First, we show that the algorithm is correct. Each edge that is not an "entrance" edge is traversed twice: the first time is when a node is named as VISITED and the second when we leave the node and never return back. Each other node is clearly traversed twice. Note that we finish the traversal at the node that we started it.

Any node keeps a label (boolean variable) that denotes if it has ever been visited or not. Also, a node keeps a reference to its "entrance" node. Finally, a node u needs to keep track of the incident edges that it has traversed (from this node to a neighboring node). This can be done by storing the next edge that need be traversed, assuming that an ordering has been defined over the neighboring nodes.

We give the pseudo-code. Each node stores three extra variables: a boolean flag $\mathsf{VISITED}(v)$, a node reference $\mathsf{ENTRANCE}(v)$ and an iterator $\mathsf{EDGE\_IT}(v)$

over incident edges. For all these extra variables we assume that there is a mechanism for setting and getting the corresponding values (in constant time). Method NextNeighbor(G,v) returns that neighbor u of v that will be next visited (that is, the corresponding edge has not been traversed from v to u and u is not the "entrance" node of v) or NULL if no such node can be visited.

```
Algorithm Traverse(G):
  v ← G.aVertex()
  VISITED(v) ← true
  ENTRANCE(v) ← NULL
  u ← NextNeighbor(G,v)
  DONE ← false
  while ¬ DONE do
    if ¬ VISITED(u) then
      VISITED(u) ← true
      ENTRANCE(u) ← e
    w ← NextNeighbor(G,u)
    if w = NULL then
      if u = v then DONE ← true
      else
          Report(u,ENTRANCE(u))
          u ← ENTRANCE(u)
    else
      Report(u,w)
      u ← w

Algorithm NextNeighbor(G,v):
  if EDGE_IT(v).hasNext() then
    o ←EDGE_IT(v).nextObject()
    u ← G.opposite(v,o)
    if u = ENTRANCE(v) then
      return NextNeighbor(v)
    else return u
  else return NULL
```

The running time of the algorithm is $O(n+m)$. Each edge is clearly "traversed" (reported) twice and at each node, the decision about what is the next node to be visited, is made in constant time.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-6.19

First, even if it is not asked, we show that a connected directed graph has an Euler tour, if and only if, each vertex has the same in-degree and out-degree (degree property from now on).

If the graph has an Euler tour, then by following that tour, it is always able to leave any vertex that we visit. Thus, the in-degree of each vertex equals its out-degree.

Assume now that each vertex has the same in-degree and out-degree. we can construct an Euler tour using the following precedure. Start at any node and keep traversing edges until you reach the same vertex (the graph is connected and by the degree property we are always able to return at this node). If this cycle (not necessary simple) is a tour (contains all edges) we are done. Otherwise, delete the traversed edges from the graph and starting by any vertex of the cycle that has non-zero degree, find (by traversing edges) another cycle (observe that the degree property holds even after the edges' removal). Remove the traversed edges and continue until no edges are left in the graph. All the cycles that where discovered can be conbined to give us an Euler tour: just keep track of the starting vertex of each cycle and insert this cycle into the previous cycle.

The last procedure corresponds to a well defined linear time algorithm for finding an Euler tour. However, we give another algorithm that, using a DFS traversal over the graph, constructs an Euler tour in a systematic way.

The algorithm is very similar to the one that traverses a connected undirected graph such that each edge is traversed exactly twice (once for each direction) (see problem 2 of homework 7a - Collaborative). there by marking the entrance edge we had been able to succesfully (that is, so that no edges were left untouched) leave a vertex and never visit it again.

This, however, can not be done here because an edge has only one direction. We, instead, "preprocess" the graph, by performing a DFS traversal in the graph. however, traversing edges in the opposite direction (that is, we have a DFS traversal where each edge is trvaersed in its opposite direction (backwards)). We only label the discovering edges in this DFS. Thus, when the DFS is over, each vertex will have a unique outgoing labeled edge (the one connecting this vertex with

its perent in the discovering tree of DFS). We then simulate the algorithm for homework 7a, problem 2: starting from any node, we perform a "blind" traversal, where we do not cross any label edge unless we are forced (there are no other way to leave the current vertex).

We give the pseudo-code that describe the algorithm. Instead of walking blindly, we assume that each vertex keep an iterator OUT_EDGE_IT of the outgoing incident edges and in this way edges are traversed in a systematic way (and thus, we do not have to label them). Also, each vertex stores an edge reference LAST_EDGE; it is a reference to an edge (the one labeled by the DFS traversal) that must be traversed only if no other option is available. Finally, instead of performing a DFS on graph $\overrightarrow{G}$ traversing edges backwards, we can perform a DFS at graph $\overrightarrow{G'}$, which is graph $\overrightarrow{G}$ having each edge with a reversed direction. We then reverse all edges (to switch back to the graph $\overrightarrow{G}$) but keep the labeling.

```
Algorithm EulerTour(G):
  for all vertices v do
    OUT_EDGE_IT(v) ← G.outIncidentEdges(v)
    LAST_EDGE(v) ← NULL
  let G' be the graph resulting by reversing
  the direction of each edge of G
  perform DFS to graph G' and for each vertex v store the
  corresponding unique outgoing discovering edge as LAST_EDGE(v)
  s ← G.aVertex()
  NextVertex(G,s)

Algorithm NextEdge(G,v):
  if OUT_EDGE_IT(v).hasNext() then
    e ← OUT_EDGE_IT(v).nextObject()
    if e =LAST_EDGE(v) then
      return NextEdge(G,v)
    else return e
  else return LAST_EDGE(v)

Algorithm NextVertex(G,v):
  e ← NextEdge(G,v)
  if e ≠ NULL then
    u = G.opposite(v,e)
    Report(e)
```

NextVertex(G,u)

The time complexity is clearly $O(n+m)$: the edge labeling takes linear time (using DFS in graph $\overrightarrow{G}$) and each edge is traversed exactly once and the decision about which edge to traverse next is made in constant time.

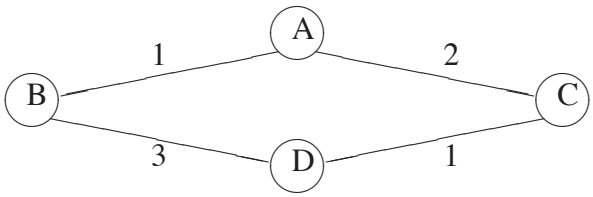
Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-7.3

The greedy algorithm presented in this problem is not guaranteed to find the shortest path between vertices in graph. This can be seen by counterexample. Consider the following weighted graph:



Suppose we wish to find the shortest path from $start = A$ to $goal = D$ and we make use of the proposed greedy strategy. Starting at A and with $path$ initialized to A, the algorithm will next place B in $path$ (because (A,B) is the minimum cost edge incident on A) and set $start$ equal to B. The lightest edge incident on $start = B$ which has not yet been traversed is (B,D). As a result, D will be appended to $path$ and $start$ will be assigned D. At this point the algorithm will terminate (since $start = goal = D$). In the end we have that $path = A,B,D$, which clearly is not the shortest path from A to D. (Note: in fact the algorithm is not guaranteed to find any path between two given nodes since it makes no provisions for backing up. In an appropriate situation, it will fail to halt.)

Algorithm Design
M. T. Goodrich and R. Tamassia
John Wiley & Sons
Solution of Exercise C-7.7

We can model this problem using a graph. We associate a vertex of the graph with each switching center and an edge of the graph with each line between two switching centers. We assign the weight of each edge to be its bandwidth. Vertices that represent switching centers that are not connected by a line do not have an edge between them.

We use the same basic idea as in Dijkstra's algorithm. We keep a variable $d[v]$ associated with each vertex v that is the bandwidth on any path from a to this vertex. We initialize the d values of all vertices to 0, except for the value of the source (the vertex corresponding to a) that is initialized to infinity. We also keep a π value associated with each vertex (that contains the predecessor vertex).

The basic subroutine will be very similar to the subroutine `Relax` in Dijkstra. Assume that we have an edge (u,v). If $\min\{d[u], w(u,v)\} > d[v]$ then we should update $d[v]$ to $\min\{d[u], w(u,v)\}$ (because the path from a to u and then to v has bandwidth $\min\{d[u], w(u,v)\}$, which is more than the one we have currently).

Algorithm `MaxBandwidth`(G, a, b):
  for `all vertices` v `in` V **do**
    $d[v] \leftarrow 0$
  $d[a] \leftarrow \infty$
  $Known \leftarrow \emptyset$
  $Unknown \leftarrow V$
  while $Unknown \neq \emptyset$ **do**
    $u \leftarrow$ `ExtractMax`$(Unknown)$
    $Known \leftarrow Known \cup \{u\}$
    for `all vertices` v `in` $Adj[u]$ **do**
      if $(min(d[u], w(u,v)) > d[v])$
        $d[v] \leftarrow min(d[u], w(u,v))$
  return `d[b]`

Here, the function `ExtractMax`$(Unknown)$ finds the node in $Unknown$ with the maximum value of d.

Complexity (this has not been asked for): If we maintain $Unknown$ as a heap, then there are $n = |V|$ `ExtractMax` operations on it and upto $m = |E|$ changes to the values of elements in the heap. We can implement a data structure which takes $O(\log n)$ time for each of these operations. Hence, the total running time is $O((n+m)\log n)$.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-7.8

Consider the weighted graph $G = (V, E)$, where V is the set of stations and E is the set of channels between the stations. Define the weight $w(e)$ of an edge $e \in E$ as the bandwidth of the corresponding channel.

Given below are two ways of solving the problem:

1. At every step of the greedy algorithm for constructing the minimum spanning tree, instead of picking the edge having the least weight, pick the edge having the greatest weight.
2. Negate all the edge-weights. Run the usual minimum spanning tree algorithm on this graph. The algorithm gives us the desired solution.

Algorithm Design
M. T. Goodrich and R. Tamassia
John Wiley & Sons
Solution of Exercise C-7.9

We will describe two solutions for the flight scheduling problem.

1. We will reduce the flight scheduling problem to the shortest paths problem. That means that we will construct an instance for the shortest paths problem, that when solved, gives us a solution for the flight scheduling problem. Given the set of airports $\mathcal{A}$ and the set of flights $\mathcal{F}$, we consider a weighted directed graph $\overrightarrow{G}$ that is constructed as follows.
   - For each airport $a_i \in \mathcal{A}$, draw a circle $circle_i$ to represent the time (24 hours).
   - For each flight $f_i \in \mathcal{F}$, find the origin airport a_o, the destination airport a_d, the departure time t_d, and the arrival time t_a. Draw a vertex v_1 on $circle_o$ marked the time t_d and also draw a vertex v_2 on $circle_d$ marked the time (t_a+c(a_d)). Draw an directed edge from v_1 to v_2 with weight t_a+c(a_d)-t_d (of course we must compute the correct weight (flight time) in case like the departure time is 10:00PM and the arrival time is 1:00AM next day). The direction of edges on a circle should be clockwise.

   Now we have a new graph like the Figure **??** below:
   Note that we have included the minimum connecting time in the total flight time, so we can only consider the modified new flights without worrying about the connecting times.
   Given graph $\overrightarrow{G}$, in order to solve the flight scheduling problem, we can just find a shortest path from the first vertex on $circle(a)$ (origin airport a) representing time t or after t to one vertex on $circle(b)$ (destination airport b) in the graph. The flights' sequence can be obtained from the shortest path from origin to destination.
   We can use Dijkstra's algorithm to find the desired shortest path (observe that $\overrightarrow{G}$ has only positive weights), but we need to slightly modify the algorithm (as it is presented in the textbook), so that it is applicable to directed graphs. The modifications should be straightforward: in the relaxation step,

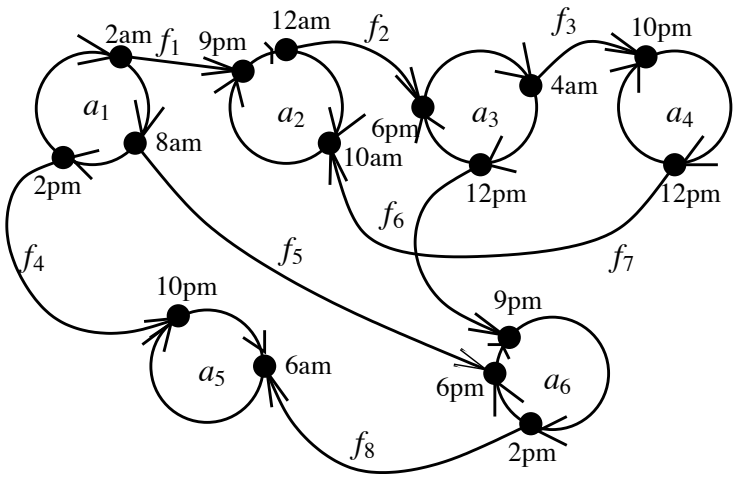


Figure 0.1: Graph used to reduce a flight scheduling problem to a shortest paths problem.

we only consider edges with orientation from the previously selected vertex to some other vertex. Finally, we need keep track (mark) the shortest path, so we include a parent-child relationship to the visited nodes (value p).

Algorithm light ched ling(a b t):
  constr ct graph $\overrightarrow{G}$
  $v \leftarrow$ rst vertex on $circle(a)$ representing time t or after t
  $D[v] \leftarrow 0$
  for each vertex $u \neq v$ of $\overrightarrow{G}$ **do**
    $D[u] \leftarrow \infty$
  let a priorit e e Q contain all the vertices of $\overrightarrow{G}$ ith D val es as ke s
  while Q is not empt **do**
    $u \leftarrow Q$.remove in()
    for each vertex z s ch that $\overrightarrow{(u,z)} \in E_{\overrightarrow{G}} \wedge z \in Q$ **do**
      if $D[u] + w(\overrightarrow{(u,z)}) < D[z]$ **then**
        $D[z] \leftarrow D[u] + w(\overrightarrow{(u,z)})$
        change to $D[z]$ the ke of vertex z in Q
        $p[z] \leftarrow u$
  $w \leftarrow$ vertex on $circle(b)$ ith minim m val e D
  return reversed path from w to v (sing val es p)

The time complexity of the algorithm essentially the time complexity of Dijkstra's algorithm (since the construction of the graph can be done in linear ($O(n+m)$) time). Using a heap as the priority queue, we achieve a total $O((N+M)\log N)$ time complexity, where N the number of vertices of $\overrightarrow{G}$ and M the number of edges of $\overrightarrow{G}$. Note that, generally, $M = O(m)$ and $N = O(m)$, so the running time of the algorithm is $O(m \log m)$.

2. The following algorithm finds the minimum travel time path from airport $a \in \mathcal{A}$ to airport $b \in \mathcal{A}$. Recall, we must depart from a at or after time t. Note, "$\oplus$" and "$\preceq$" are operations which we must implement. We define these operations as follows: If $x \oplus y = z$, then z is the point in time (with date taken into consideration) which follows x by y time units. Also, if $a \preceq b$, then a is a point in time which preceeds or equals b. These operations can be implemented in constant time.

Algorithm light ched ling($a\ b\ t$):
  initialize a set $\mathcal{P}$ to a
  initialize incoming_ ight(x) to nil for each $x \in \mathcal{A}$
  earliest_arrival_time(a)$\leftarrow t$
  for all $x \in \mathcal{A}$ s ch that $x \neq a$ **do**
    earliest_arrival_time(x)$\leftarrow \infty$
    time_can_depart$\leftarrow \infty$
  repeat
    remove from $\mathcal{P}$ airport x s ch that earliest_arrival_time(x) is soonest
    if $x \neq b$ **then**
      if $x = a$ **then**
        time_can_depart $\leftarrow t$
      else
        time_can_depart $\leftarrow$ earliest_arrival_time(x) $\oplus$ c(x)
      for each ight $f \in \mathcal{F}$ s ch that $a_1(f) = x$ **do**
        if $time\_can\_depart \preceq t_1(f) \wedge$
        $t_2(f) \preceq earliest\_arrival\_time(a_2(f))$ **then**
          earliest_arrival_time($a_2(f)$) $\leftarrow t_2(f)$
          incoming_ ight($a_2(f)$) $\leftarrow f$
          add $a_2(f)$ to $\mathcal{P}$ if it is not et there
  until $x = b$

  initialize *city* to b

initialize $flight\_sequence$ to nil
while (cit $\neq a$) **do**
  append $incoming\_flight(city)$ to $flight\_sequence$
  assign to $city$ the depart re cit of $incoming\_flight(city)$
reverse $flight\_sequence$ and o tp t

The algorithm essentially performs Dijkstra's Shortest Path Algorithm (the cities are the vertices and the flights are the edges). The only small alterations are that we restrict edge traversals (an edge can only be traversed at a certain time), we stop at a certain goal, and we output the path (a sequence of flights) to this goal. The only change of possible consequence to the time complexity is the flight sequence computation (the last portion of the algorithm). A minimum time path will certainly never contain more that m flights (since there are only m total flights). Thus, we may discover the path (tracing from b back to a) in $O(m)$ time. Reversal of this path will require $O(m)$ time as well. Thus, since the time complexity of Dijkstra's algorithm is $O(m \log n)$, the time complexity of our algorithm is $O(m \log n)$.

Note: Prior to execution of this algorithm, we may (if not given to us) divide $\mathcal{F}$ into subsets $F_1, F_2, \ldots, F_N$, where F_i contains the flights which depart from airport i. Then when we say "for each flight $f \in \mathcal{F}$ such that $a_1(f) = x$", we will not have to waste time looking through flights which do not depart from x (we will actually compute "for each flight $f \in F_x$"). This division of $\mathcal{F}$ will require $O(m)$ time (since there are m total flights), and thus will not slow down the computation of the minimum time path (which requires $O(m \log n)$ time).

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-7.10

Simply run the shortest path algorithm with $path(a) < path(b)$ if there is more gold on $path(a)$ than $path(b)$.

Algorithm Design

M. T. Goodrich and R. Tamassia

John Wiley & Sons

Solution of Exercise C-7.11

1. If $M^2(i,j) = 1$, then there is a path of length 2 (a path traversing exactly 2 edges) from vertex i to vertex j in the graph G. Alternatively, if $M^2(i,j) = 0$, then there is no such path.
2. Similarly, if $M^4(i,j) = 1$, then there exists a path of length 4 from v_i to v_j, otherwise no such path exists. The situation with M^5 is analogous to that of M^4 and M^2. In general, M^p gives us all the vertex pairs of G which are connected by paths of length p.
3. if $M^2(i,j) = k$, then we can conclude that the shortest path connecting vertices i and j of length ≤ 2 has weight k. That is, if $k = \infty$, then i and j are not connected with a path of length at most 2, and, if $k \neq \infty$, then they are connected with a path of length at most 2 that has weight k and is a shortest path of length ≤ 2.