**1)** Show that the greedy approach always ﬁnds an optimal solution for the Change problem when the coins are in the denominations D0, D1, D2,... , Di for some integers i > 0 and D > 1.

Let Sg be the greedy solution, written in polynomial order Sg = giDi + gi-1Di-1 + … + g1D + g0, and let Sh be a non-greedy solution Sh = hiDi + hi-1Di-1 + … + h1D + h0. (All the numbers involved are positive integers.)

We are going to show that  , which means that Sg is the optimal solution.

Being solutions, both sums are equal to the required number of units of change: S = Sg = Sh.

Since Sg is greedy, all the coefficients gj are in the interval [0…D-1].

Since Sh is not greedy, let ***m*** be the first index (from left to right) where Sh does ***not*** make the greedy choice, i.e. gm > hm. By subtraction we get:

Sg – Sh = (gm – hm)Dm + (gm-1 – hm-1)Dm-1 + … + (g1 – h1)D + (g0 – h0) = 0

Since all the coefficients greater than ***m*** are identical, it remains to prove that

(gm – hm) + (gm-1 – hm-1) + … + (g1 – h1) + (g0 – h0) < 0

For concision, we denote (gj – hj) by pj. Remember that gm > hm, so pm > 0. (But we don’t know the signs of the other differences.) We now have pmDm + pm-1Dm-1 + … + p1D + p0 = 0, and we need to show that pm + pm-1 + … + p1 + p0 < 0.

Apply induction on m:

Base case: When m = 1, we have p1D + p0 = n0 → p1(D-1) + p1 + p0= n0 → p1 + p0 < n0

General case: We assume that the result holds for ***m-1***, and we prove it for ***m***. We start from pmDm + pm-1Dm-1 + … + p1D + p0 = 0, and divide by D to get

pmDm-1 + pm-1Dm-2 + … + p1 + p0/D = 0.

According to the induction hypothesis, we have pm + pm-1 + … + p1 + p0/D < 0 (\*)

Consider the 2 possible cases for p0:

1. If p0 < 0, since D > 1, (\*) implies pm + pm-1 + … + p1 + p0 < 0
2. If p0 ≥ 0, it is time to remember that p0 = g0 – h0 , and, as noted at the beginning, all greedy coefficients gj, including g0, are in the interval [0…D-1]. Since h0 ≥ 0, we have that also p0 = [0…D-1]. This makes it impossible for p0/D in eq. (\*) to be an integer unless p0 = 0, in which case (\*) trivially implies pm + pm-1 + … + p1 + p0 < 0

The induction conclusion was proven, q.e.d.

**2)** 1. Begin with Y = { v1 }, F = { }.

2. Nearest vertex to Y is v4 (distance 14), so add v4 to Y and (v1, v4) to F.

3. Nearest vertex to Y is v8 (distance 3), so add v8 to Y and (v4, v8) to F.

4. Nearest vertex to Y is v9 (distance 4), so add v9 to Y and (v8, v9) to F.

5. Nearest vertex to Y is v5 (distance 10), so add v5 to Y and (v4, v5) to F.

6. Nearest vertex to Y is v10 (distance 12), so add v10 to Y and (v9, v10) to F.

7. Nearest vertex to Y is v6 (distance 6), so add v6 to Y and (v6, v10) to F.

8. Nearest vertex to Y is v3 (distance 18), so add v3 to Y and (v3, v4) to F.

9. Nearest vertex to Y is v7 (distance 5), so add v7 to Y and (v3, v7) to F.

10. Nearest vertex to Y is v2 (distance 32), so add v2 to Y and (v1, v2) to F.

11. Now Y == V, so instance is solved. Spanning tree is given by edges in

F = { (v1, v4), (v4, v8), (v8, v9), (v4, v5), (v9, v10), (v6, v10), (v3, v4), (v3, v7), (v1, v2) }

**3)** a) 1. Beginning with vertex v4, each table shows the set of edges over which we search for the min at each step (indicated by shaded squares). Initially, Y = { v4 } and F = { }.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 0 | INF | 72 | 50 | 90 | 35 |
| 2 | INF | 0 | 71 | 70 | 73 | 75 |
| 3 | 72 | 71 | 0 | INF | 77 | 90 |
| 4 | 50 | 70 | INF | 0 | 60 | 40 |
| 5 | 90 | 73 | 77 | 60 | 0 | 80 |
| 6 | 35 | 75 | 90 | 40 | 80 | 0 |

2. Min edge is 40 from v4 to v6, so we add v6 to Y and (v4, v6) to F.

3. Pick greedily again with Y = { v4, v6 }.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 0 | INF | 72 | 50 | 90 | 35 |
| 2 | INF | 0 | 71 | 70 | 73 | 75 |
| 3 | 72 | 71 | 0 | INF | 77 | 90 |
| 4 | 50 | 70 | INF | 0 | 60 | 40 |
| 5 | 90 | 73 | 77 | 60 | 0 | 80 |
| 6 | 35 | 75 | 90 | 40 | 80 | 0 |

4. Min edge is 35 from v1 to v6, so we add v1 to Y and (v1, v6) to F.

5. Pick greedily again with Y = { v1, v4, v6 } and F = { (v4, v6), (v1, v6) }.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 0 | INF | 72 | 50 | 90 | 35 |
| 2 | INF | 0 | 71 | 70 | 73 | 75 |
| 3 | 72 | 71 | 0 | INF | 77 | 90 |
| 4 | 50 | 70 | INF | 0 | 60 | 40 |
| 5 | 90 | 73 | 77 | 60 | 0 | 80 |
| 6 | 35 | 75 | 90 | 40 | 80 | 0 |

6. Min edge is 60 from v4 to v5, so we add v5 to Y and (v4, v5) to F.

7. Pick greedily again with Y = { v1, v4, v5, v6 } and F = { (v4, v6), (v1, v6), (v4, v5) }.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 0 | INF | 72 | 50 | 90 | 35 |
| 2 | INF | 0 | 71 | 70 | 73 | 75 |
| 3 | 72 | 71 | 0 | INF | 77 | 90 |
| 4 | 50 | 70 | INF | 0 | 60 | 40 |
| 5 | 90 | 73 | 77 | 60 | 0 | 80 |
| 6 | 35 | 75 | 90 | 40 | 80 | 0 |

8. Min edge is 70 from v2 to v4, so we add v2 to Y and (v2, v4) to F.

9. Pick greedily again with Y = { v1, v2, v4, v5, v6 } and F = { (v4, v6), (v1, v6), (v4, v5), (v2, v4) }.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 0 | INF | 72 | 50 | 90 | 35 |
| 2 | INF | 0 | 71 | 70 | 73 | 75 |
| 3 | 72 | 71 | 0 | INF | 77 | 90 |
| 4 | 50 | 70 | INF | 0 | 60 | 40 |
| 5 | 90 | 73 | 77 | 60 | 0 | 80 |
| 6 | 35 | 75 | 90 | 40 | 80 | 0 |

10. Min edge is 71 from v2 to v3, so we add v3 to Y and (v2, v3) to F.

11. Now Y == V, so problem is solved.

b) Minimum spanning tree is given by F = { { (v4, v6), (v1, v6), (v4, v5), (v2, v4), (v2, v3) }.

c) Cost of spanning tree is 40 + 35 + 60 + 70 + 71 = 276.

**4)** Graph G containing more than one spanning tree is given by adjacency matrix:

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 5 | 9 | INF |
| 5 | 0 | 6 | 6 |
| 9 | 6 | 0 | 7 |
| INF | 6 | 7 | 0 |

Two minimum spanning trees are given by F = { (v1, v2), (v2, v4), (v3, v4) } and F = { (v1, v2), (v2, v3), (v3, v4) }, each with cost 18.

**5)** Solutions will vary.

**6)** Modify Prim’s algorithm (Algorithm 4.1) to check if an undirected, weighted

graph is connected. Analyze your algorithm and show the results using order

notation.

Boolean connected (int *n*, const number *w*[ ] [ ]) {

index *i*, *vnear*;

number min;

edge *e*;

index nearest [2, … *n*]

number distance [2, … *n*];

boolean connected = true;

for (*i* = 2; *i* < = *n*; *i*++){

nearest [*i*] = 1

distance [*i*] = *w* [1] [*i*];

}

repeat (*n* − 1 times){

min = infinity;

for (*i* = 2; *i* < = *n*; *i* + +)

if (0 < = distance [*i*] < min) {

min = distance [*i*];

*vnear* = *i*;

}

if (min == infinity) {

connected = false;

break;

}

distance [*vnear*] = −1;

for (*i* = 2; *i* < = *n*; *i*++)

if (*w* [*i*] [*vnear*] < distance [*i*]) {

distance [*i*] = w [*i*] [*vnear*];

nearest [*i*] = *vnear*;

}

}

return connected;

}

Complexity analysis: As in the case of the original Prim algorithm, T(n) = (n-1)\*2\*(n-1) → (*n*2).

**7)** Use Kruskal’s algorithm (Algorithm 4.2) to ﬁnd a minimum spanning tree for the graph in Exercise 2. Show the actions step by step.

(*v*4, *v*8) → 3 → Accepted

(*v*8, *v*9) → 4 → Accepted

(*v*3, *v*7) → 5 → Accepted

(*v*6, *v*10) →6 → Accepted

(*v*4, *v*5) → 10 → Accepted

(*v*9, *v*10) → 12 → Accepted

(*v*1, *v*4) → 17 → Accepted

(*v*3, *v*4) → 18 → Accepted

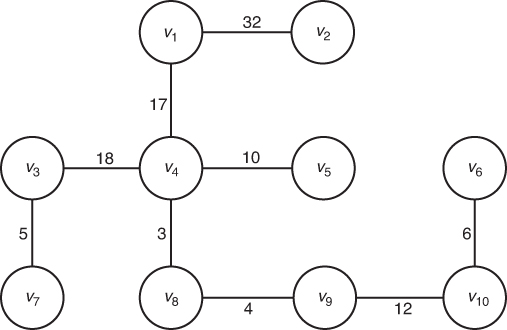
(*v*5, *v*9) → 25 → Rejected

(*v*5, *v*6) → 28 → Rejected

(*v*1, *v*2) → 32 → Accepted

(*v*2, *v*5) → 45 → Rejected

(*v*7, *v*8) → 59 → Rejected



**8)** Implement Kruskal’s algorithm (Algorithm 4.2) on your system, and study its performance using diﬀerent graphs.

Solutions and performance will vary.

**9)** It is not possible for a minimum spanning tree to have a cycle since all trees are acyclic by definition.

**10)** Assume that in a network of computers any two computers can be linked. Given a cost estimate for each possible link, should Algorithm 4.1(Prim’s algorithm) or Algorithm 4.2 (Kruskal’s algorithm) be used? Justify your answer.

Prim’s algorithm is(*n*2), and Kruskal’s algorithm is(*m∙lgm*), where m is the number of edges in the graph. Since our graph is completely connected, m = n(n-1)/2. Kruskal’s algorithm in terms of n is therefore(*n2∙lgn*), slightly less efficient than Prim’s.

**11)** Apply Lemma 4.2 to complete the proof of Theorem 4.2.

*Let G* = (*V*, *E*) be a graph with a set of nodes V and a set of edges E. Let n be the number of nodes.

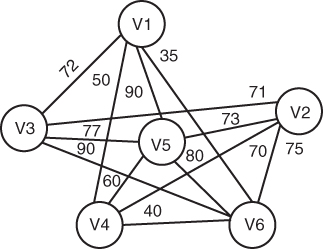
We use induction by n:

* Base case: If n = 1, the graph has only one node and no edges (no self-edges), so the while loop in Kruskal’s algorithm does not execute at all. The minimal spanning tree (MST) has just a node and no edges, which is correct.
* Induction step: We assume that Kruskal’s algorithm produces a MST for any graph of n-1 nodes. In a graph with n nodes, we apply Kruskal’s algorithm and focus our attention on the set of edges Fn-2 that is obtained in the next-to-last iteration. According to the hypothesis, Fn-2 is a MST for the sub-graph with n-1 nodes. Is Fn-2 also promising? Yes! (Reasoning by contradiction, if it were not promising, there would be no edge from the n-1-node subgraph to node n that would create a MST for the n-node graph. But such an edge e must be part of any MST. If we remove e from the MST, we get a smaller MST for the n-1-node subgraph – contradiction.) We can apply the lemma, which says that Fn-1 is promising in G, therefore it is MST for G. q.e.d.

**4.2**

**12)** Use Dijkstra’s algorithm to find the shortest path from vertex *v*5 to all other vertices for the graph represented by the array shown in the table. Show the actions step by step.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 |
| 1 | 0 | ∞ | 72 | 50 | 90 | 35 |
| 2 | ∞ | 0 | 71 | 70 | 73 | 75 |
| 3 | 72 | 71 | 0 | ∞ | 77 | 90 |
| 4 | 50 | 70 | ∞ | 0 | 60 | 40 |
| 5 | 90 | 73 | 77 | 60 | 0 | 80 |
| 6 | 35 | 75 | 90 | 40 | 80 | 0 |



Solution: From the above table, form the graph:

Use Dijkstra’s algorithm to find the shortest path from vertex *v*5:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Unvisited | Visited | 5 | 1 | 2 | 3 | 4 | 6 |
| {1, 2, 3, 4, 5, 6} |  | (0, −) | (∞, −) | (∞, −) | (∞, −) | (∞, −) | (∞, −) |
|  |  |  |  |  |  | ⇓ |  |
| {1, 2, 3, 4, 5, 6} | {5} | ( ) | (90, 5) | (73, 5) | (77, 5) | (60, 5) | (80, 5) |
|  |  |  |  | ⇓ |  |  |  |
| {1, 2, 3, 6} | {4, 5} |  | (90, 5) | (73, 5) | (77, 5) |  | (80, 5) |
|  |  |  |  |  | ⇓ |  |  |
| {1, 3, 6} | {2, 4, 5} |  | (90, 5) |  | (77, 5) |  | (80, 5) |
|  |  |  |  |  |  |  | ⇓ |
| {1, 6} | {2, 3, 4, 5} |  | (90, 5) |  |  |  | (80, 5) |
|  |  |  | ⇓ |  |  |  |  |
| {1} | {1, 2, 3, 4, 5, 6} |  | (90, 5) |  |  |  |  |
| {} | {1, 2, 3, 4, 5, 6} |  |  |  |  |  |  |

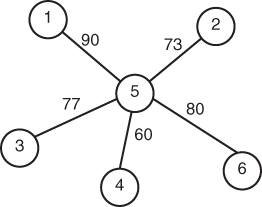
\* Unvisited nodes indicate that the node has not yet processed/reached.

\* 5 being the initialized vertex, it is chosen first as the visited node.

\* Using the algorithm, the cost from *v*5 to all searchable nodes is calculated.

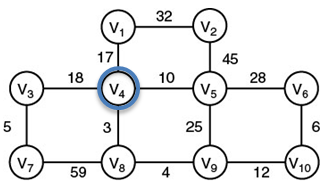
\* Then, the neighbor vertex in the lowest path is chosen as the next vertex [Here, vertex 4 with cost 60 is chosen].

\* Then, using the conditional statements,



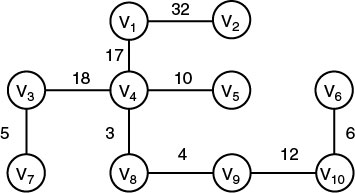
From the table, the following graph is drawn:

**13)** Use Dijkstra’s algorithm to find the shortest paths from vertex *v*4 to all other vertices of the graph shown in the figure. Show the actions step by step. Assume that each undirected edge represents two directed edges with the same weight.



|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Unvisited | Visited | V4 | V1 | V2 | V3 | V5 | V6 | V7 | V8 | V9 | V10 |
| {1, 2, 3, 5, 6, 7, 8, 9, 10} | {4} | (0, ∞) | (17, 4) | (∞, −) | (18, 4) | (10, 4) | (∞, −) | (∞, −) | (3, 4) | (∞, −) | (∞, −) |
|  |  |  |  |  |  |  |  |  |  | ⇓ |  |
| {1, 2, 3, 5, 6, 7, 9, 10} | {4, 8} |  | (17, 4) | (∞, −) | (18, 4) | (10, 4) | (∞, −) | (62, 8) |  | (7, 8) | (∞, −) |
|  |  |  |  |  |  | ⇓ |  |  |  |  |  |
| {1, 2, 3, 5, 6, 7, 10} | {4, 8, 9} |  | (17, 4) | (∞, −) | (18, 4) | (10, 4) | (∞, −) | (62, 8) |  |  | (∞, −) |
|  |  |  | ⇓ |  |  |  |  |  |  |  |  |
| {1, 2, 3, 6, 7, 10} | {4, 5, 8, 9} |  | (17, 4) | (55, 5) | (18, 4) |  | (38, 5) | (62, 8) |  |  | (19, 9) |
|  |  |  |  |  | ⇓ |  |  |  |  |  |  |
| {2, 3, 6, 7, 10} | {1, 4, 5, 8, 9} |  | (17, 4) | (49, 1) | (18, 4) |  | (38, 5) | (23, 3) |  |  | (19, 9) |
|  |  |  |  |  |  |  |  |  |  |  | ⇓ |
| {2, 6, 7, 10} | {1, 3, 4, 5, 8, 9} |  |  | (49, 1) |  |  | (38, 5) | (23, 3) |  |  | (19, 9) |
|  |  |  |  |  |  |  |  | ⇓ |  |  |  |
| {2, 6, 7} | {1, 3, 4, 5, 8, 9, 10} |  |  | (49, 1) |  |  | (25, 10) | (23, 3) |  |  |  |
|  |  |  |  |  |  |  | ⇓ |  |  |  |  |
| {2, 6,} | {1, 3, 4, 5, 7, 8, 9, 10} |  |  | (49, 1) |  |  | (25, 10) |  |  |  |  |
|  |  |  |  | ⇓ |  |  |  |  |  |  |  |
| {2} | {1, 3, 4, 5, 6, 7, 8, 9, 10} |  |  | (49, 1) |  |  |  |  |  |  |  |
| { } | {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} |  |  |  |  |  |  |  |  |  |  |

Solution graph:



**14)** Solutions and performance will vary.

**15)** Modify Dijkstra's algorithm (Algorithm 4.3) so that it computes the lengths of the shortest paths. Analyze the modified algorithm and show the results using order notation.

The solution requires only a slight change in Algorithm 4.3, based on the observation that, until ***vnear*** is added to Y by setting its ***length*** to -1 (last line), the array ***length*** does store the length of the shortest path. All we need to do is to save that length in another array, say ***report*** which the function returns. The function will have one more argument:

void dijkstra(int n, const number W[][], number ***report[]***, set\_of\_edges& F)

**16)** Modify Dijkstra’s algorithm (Algorithm 4.3) so that it checks if a directed graph has a cycle. Analyze your algorithm and show the results using order notation.

The existence of a cycle has nothing to do with the metric properties of the graph, so for simplicity we can replace all the weights with the value 1.

For each vertex v:

Run Dijkstra (Algorithm 4.3) on the new graph //(n2)

Check for “back edge” from all other nodes to v //(n)

If back edge exists

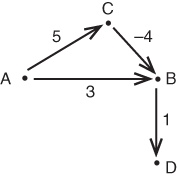
cycle was found

exit

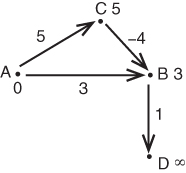
Complexity analysis: The outer for loop executes n times →(n3)

**17)** Can Dijkstra’s algorithm be used to find the shortest paths in a graph with some negative weights? Justify your answer.

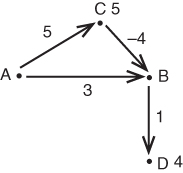
Dijkstra’s algorithm cannot in general handle negative weights. Here’s a counterexample:



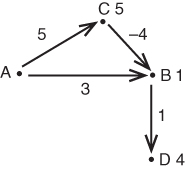
First, source A is visited, and the distances to B and C are calculated



B has the shortest distance, so it’s added to the set of visited nodes. The distances are updated:



D has the shortest distance, so we visit it and update distances:



The only vertex now left in the queue is C, so we visit it, with a distance of 5, and the algorithm terminates. The path A → C → B → D has total length 5 + (–4) + 1 = 2 < 4; thus, the length of the shortest path from source A to D was not correctly evaluated.

End of example

The best-known algorithm to find single source shortest path in a directed graph with negative weights is Bellman–Ford algorithm. It is possible to adapt Dijkstra’s algorithm to handle negative weights by combining it with Bellman–Ford (Johnson’s algorithm).

**18)** Use induction to prove the correctness of Dijkstra’s algorithm.

Theorem (the correctness of Dijkstra’s algorithm 4.3):

When a node joins the set of edges F, the distance in its label is the length of the shortest path.

Proof: At each iteration of the ***repeat*** loop, the nodes are partitioned into subsets F and V – F.

We prove by induction on the size of F that the label for each node i ∈ F is the shortest distance from node 1.

Base case: when |F| = 1, the only node in F is 1. It is correctly labeled with a distance 0.

Inductive step: Assume for |F| = k and prove for |F| = k + 1. Suppose that node i with d(i) = min{d(j): j ∈ V – F} is added to F, and assume by contradiction that its label d(i) is not the shortest path label. Then, there should be some nodes in V – F along the shortest path from 1 to i.

Let j be the first node in V – F on the shortest path from 1 to i. The length of shortest path from 1 to i is d(i) + Cji. Since d(i) is not the shortest label, d(j) + Cji < d(i), and since Cji is positive, we have d(j)<d(i). This contradicts that d(i) is the minimum of d(j) ∈ V – F.

**4.3**

**19)** Consider the following jobs and service times. Use the algorithm in Section 4.3.1 to minimize the total amount of tunes spent in the system.

|  |  |
| --- | --- |
| Job | Service Time |
| 1 | 7 |
| 2 | 3 |
| 3 | 10 |
| 4 | 5 |

Solution:

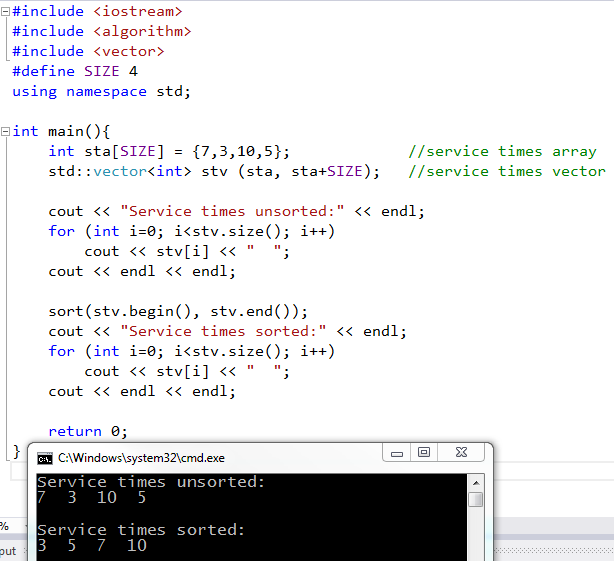
Step1: Arrange and sort the jobs by service time in nondecreasing order.



|  |  |
| --- | --- |
| Schedule | Total Time in the System |
| [*t*2, *t*4, *t*1, *t*3] | 3 + (3 + 5) + (3 + 5 + 7) + (3 + 5 + 7 + 10) = 51 |

Schedule [2, 4, 1, 3] is optional with a total time of 32.

**20)** Implement the algorithm in Section 4.3.1 on your system, and run it on the instance in Exercise 19.



**21)** Write an algorithm for the generalization of the single-server scheduling problem to the multiple-server scheduling problem in Section 4.3.1. Analyze your algorithm and show the results using order notation.

Solution: The algorithm is based on the explanations at the end of Section 4.3.2.

Let ***m*** be the number of servers, and ***n*** the number of jobs. The servers are identical for our purposes, so their order does not matter.

Algorithm:

Sort the ***n*** jobs by service time in nondecreasing order

Set variable ***job\_counter*** to 0

For i from 0 to infinity:

For j from 1 to m:

Schedule job j + i\*m on server j

Increment ***job\_counter***

If (***job\_counter*** == n) exit //All jobs are scheduled

As an example, consider *m* = 2 servers and the following jobs:

|  |  |
| --- | --- |
| Job | Service Time |
| 1 | 7 |
| 2 | 3 |
| 3 | 10 |
| 4 | 5 |

Sort the jobs in nondecreasing order, *t*2 = 3, *t*4 = 5, *t*1 = 7, *t*3 = 10.

|  |  |
| --- | --- |
| Server 1 | Server 2 |
| *t*2, *t*1 | *t*4, *t*3 |
| = 3 + (3 + 7) | = (5) + (5 + 10) |
| = 13 | = 20 |

Total time spent in the system = 33 [*t*2, *t*4, *t*1, *t*3].

**22)** Consider the following jobs, deadlines, and profits, and use the scheduling with deadlines algorithm (Algorithm 4.4) to maximize the total profit.

|  |  |  |
| --- | --- | --- |
| Job | Deadline | Profit |
| 1 | 2 | 40 |
| 2 | 4 | 15 |
| 3 | 3 | 60 |
| 4 | 2 | 20 |
| 5 | 3 | 10 |
| 6 | 1 | 45 |
| 7 | 1 | 55 |

Algorithm 4.4 does the following:

1. Sort the jobs in non-increasing order of profits:

|  |  |  |
| --- | --- | --- |
| Job | Deadline | Profit |
| 3 | 3 | 60 |
| 7 | 1 | 55 |
| 6 | 1 | 45 |
| 1 | 2 | 40 |
| 4 | 2 | 20 |
| 2 | 4 | 15 |
| 5 | 3 | 10 |

1) *J* is set to [3].

2) *K* is set to [3,7] and is determined to be feasible, so *J* is set to [3,7].

3) *K* is set to [3,7,6] and is determined to be not feasible, so *J* remains [3, 7].

4) *K* is set to [3,7,1] and is determined to be feasible, so *J* is set to [3,7,1].

5) *K* is set to [3,7,1,4] and is determined to be not feasible, so *J* remains [3,7,1].

6) *K* is set to [3,7,1,2] and is determined to be feasible, so *J* is set to [3,7,1,2].

*7) K* is set to [3,7,1,2,5] and is determined to be not feasible, so *J* remains [3,7,1,2].

The maximum profit for schedule [3,7,1,2] is 170.

**23)** Consider procedure schedule in the Scheduling with Deadlines algorithm (Algorithm 4.4). Let d be the maximum of the deadlines for n jobs. Modify the procedure so that it adds a job as late as possible to the schedule being built, but no later than its deadline. Do this by initializing d+1 disjoint sets, containing the integers 0, 1, ..., d. Let small(S) be the smallest member of set S. When a job is scheduled, ﬁnd the set S containing the minimum of its deadline and n. If small(S) = 0, reject the job. Otherwise, schedule it at time small(S), and merge S with the set containing small(S)−1. Assuming we use Disjoint Set Data Structure III in Appendix C, show that this version is θ(nlgm), where m is the minimum of d and n.

The modified scheduling procedure is:

void schedule\_just\_in\_time ( int n, int d,

const int deadline[],

sequence & J)

{

index i, minim, set\_ptr;

int smalley;

J = []; //empty sequence

universe U; //see Disjoint Set Data Structure III in App.C

for (i=0; i<=d; i++)

makeset(i);

for (i=1; i <= n; i++){

minim = min(n, deadline[i]);

set\_ptr = find(minim);

smalley = small(set\_ptr);

if (smalley > 0){

add the pair (i, smalley) to J;

merge(set\_ptr, find(smalley-1));

}

}

}

Complexity analysis: The main for loop above executes n times, and in it the dominant function is find(), whose complexity in the worst case is the maximum height of the trees used to represent the sets in Disjoint Set Data Structure III. The maximum number of elements in any of those sets is indeed bounded by both n and d:

* If n >= d, the total number of elements in all sets is d+1, so the largest tree to be searched will have at most d+1-1 = d elements (at the previous step).
* If n < d, remember that the algorithm always calculates min(n, deadline[i]), and the scheduled time smalley is possibly even less than that minimum, so the size of the largest tree is also bounded by n.

Denote m = min(d, n). The height of a balanced tree is logarithmic (lgm), so overall we have (n∙lgm) q.e.d.

As an example, let’s apply the algorithm to the jobs from Ex.22, sorted by profit:

3 3 60

7 1 55

6 1 45

1 2 40

4 2 20

2 4 15

5 3 10

We have d = 4 and n = 7. Create these 5 sets:

{0} {1} {2} {3} {4}

Scheduling job 3: min(3,7) = 3 → go to set {3}, whose small is 3 → schedule at t=3, merge {3} and {2} into {2,3}. The sets are:

{0} {1} {2, 3} {4}

Scheduling job 7: min(1,7) = 1 → go to set {1}, whose small is 1 → schedule at t=1, merge {1} and {0} into {0,1}. The sets are:

{0, 1} {2, 3} {4}

Scheduling job 6: min(1,7) = 1 → go to set {0,1}, whose small is 0 → cannot schedule

Scheduling job 1: min(2,7) = 2 → go to set {2,3}, whose small is 2 → schedule at t=2, merge {2,3} and {0,1} into {0,1,2,3}. The sets are:

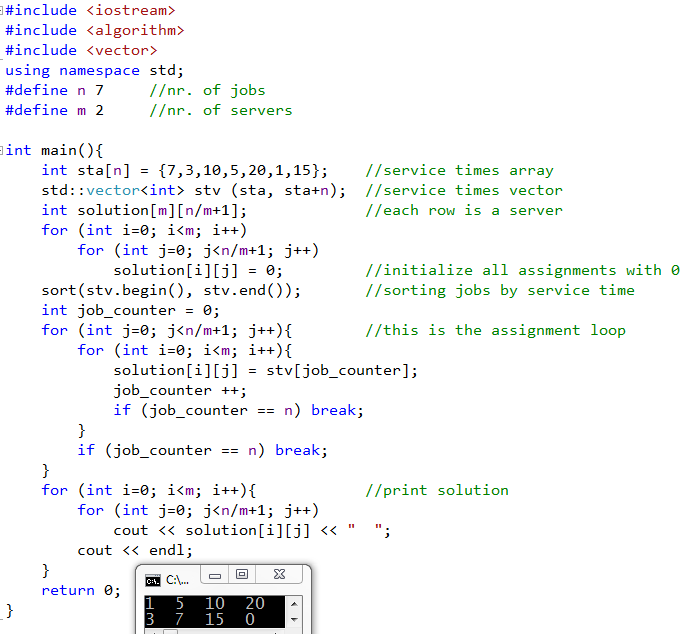
{0, 1, 2, 3} {4}

Scheduling job 4: cannot schedule

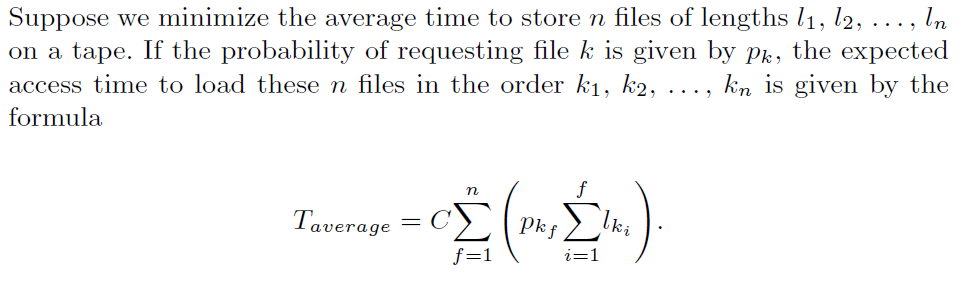
Scheduling job 2: schedule at t=4 and merge everything in to {0,1,2,3,4}

Scheduling job 5: cannot schedule

**24)** Implement the algorithm developed in Exercise 21.



**25)**



The constant C represents parameters such as the speed of the drive and the

recording density.

a) In what order should a greedy approach store these files to guarantee minimum average access time?

Solution: Consider the following example:

|  |  |  |
| --- | --- | --- |
|  | *l* | *p* |
| *f*1 | 5 | 0.4 |
| *f*2 | 4 | 0.5 |
| *f*3 | 3 | 0.1 |

Sort the files from smallest ratio of length over access probability to largest ratio of length over access probability:



The files are stored in the tape in the following order: *f*2, *f*1, *f*3, hence the average access time is minimized.

b) Write the algorithm to store the files, analyze your algorithm, and show the results using order notation.

Solution: The dominant term is sorting n ratios, which can be done in (n·lgn), e.g. with Mergesort or Quicksort.

**4.4**

**26)** A: 00

B: 011

I: 10

M: 111

S: 110

X: 0101

Z: 0100

**27)** c: 001

e: 01

i: 111

r: 100

s: 110

t: 000

x: 101

**28)** Decode each bit string using the binary code in Exercise26.

From the answer to Exercise 26 we have:

*A* − 00

*B* − 011

*I* − 10

*M* − 111

*S* − 110

*X* − 0101

*Z* − 0100

a)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 011 | 00 | 0101 | 0101 | 0 |
| B | A | X | X |  |

b)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 10 | 00 | 10 | 00 | 0101 | 0 |
| I | A | I | A | X |  |

c)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 111 | 00 | 10 | 011 | 110 | 1 |
| M | A | I | B | S |  |

d)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 10 | 00 | 0100 | 111 | 00 |
| I | A | Z | M | A |

Note: When there are unused bits at the end of a string, they are part of the next character, which has not completely arrived yet.

**29)** Using the code generated in Exercise 27:

a) rise -> 10011111001

b) exit -> 01101111000

c) text -> 00001101000

d) exercise -> 011010110000111111001

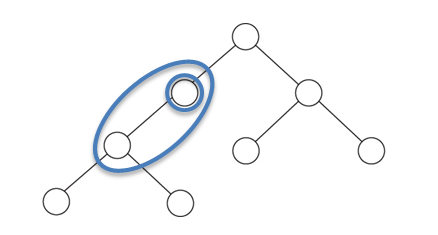
**30)** a: 00, b: 01, c: 101, d: 110, e: 111, so x = 1, y = 1, and z = 1.

**31)** Implement Huffman's algorithm, and run it on the problem instances of Exercises 26 and 27.

Solutions will vary.

**32)** Show that a binary tree corresponding to an optimal binary prefix code must be full. A full binary tree is a binary tree such that every node is either a leaf or it has two children.

Proof by contradiction: Assuming that a node in an optimal tree T has only 1 child, the entire sub-tree rooted at the child can be shifted up by collapsing the parent and the child:



The resulting tree T’ has lower cost, because all the depths in the shifted sub-tree have decreased by 1. This contradicts the optimality of T.

**33)** Prove that for an optimal binary prefix code, if the characters are ordered so that their frequencies are non-increasing, then their codeword lengths are non-decreasing.

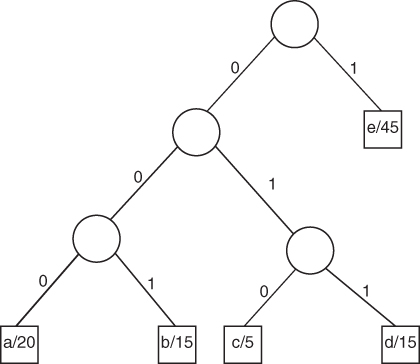
Proof by contradiction: Let the characters c1, c2, … cn be ordered in non-increasing order of frequencies:

f1 ≥ f2 ≥ … ≥ fn. Assume the codeword lengths L1, L2, …, Ln are not non-decreasing, meaning that there exists (at least) an index i such that Li > Li+1. This means that ci is placed deeper in the tree than ci+1. Since fi ≥ fi+1, we have two cases:

* fi > fi+1 →By exchanging ci and ci+1,we obtain a tree whose cost is (Li - Li+1)( fi - fi+1) smaller, which contradicts optimality.
* fi = fi+1 →We can reasonably assume that in the priority queue used in the Huffman algorithm, fi+1 is placed closer to the front than fi, so this case cannot occur.

Example:

*A* = { *e*/45, *a*/20, *b*/15, *d*/15, *c*/5 } is the non-increasing frequency order.



*e → 1, a* → 000, *b* → 001, *d* → 011, *c* → 010, so the codewords are non-decreasing.

**34)** Given the binary tree corresponding to a binary prefix code, write an algorithm that determines the codewords for all the characters. Determine the time complexity of your algorithm.

We perform a tree traversal (any traversal will do, e.g. BFS, DFS), while keeping track of the sequence of 0s and 1s that leads to the current node. The sequence is stored in a list LZO (List of Zeroes and Ones)

Pseudocode for recursive infix traversal, i.e. Center – Left – Right:

void codewords(node, LZO){

add the node’s zero or one to LZO

if node is leaf

print node\_symbol, LZO

else

codeword(left\_child, LZO)

codeword(roght\_child, LZO)

}

Complexity analysis: (*n*), because at each node in the tree we only have to update the list of 0s and 1s twice: once when we enter the node from above, and once when we leave upwards.

**4.5**

**35)** Problem: Given a set S of n items each with a weight w and a value v, select a subset of the items such that the total value of the items in the subset is maximized and the total weight of the items in the subset does not exceed a given threshold W.

Input: A set S of items indexed from 1 to n, each of which has value vi and weight wi, and an integer total weight capacity W.

Output: The value of the maximum-valued set of items in S whose total weight does not exceed W.

int knapsack(int w[1…n], int v[1…n], int W) {

index i, j;

int V[0…n][0…W];

for(i=0; i<=n; i++)

V[i][0] = 0;

for(j=0; j<=W; j++)

V[0][j] = 0;

for(i=1; i<=n; i++) {

for(j=1; j<=W; j++) {

if(w[i] > j)

V[i][j] = V[i-1][j];

else

V[i][j] = max(V[i-1][j], v[i] + V[i-1][j – w[i]);

}

}

return V[n][W];

}

**36)** Use a greedy approach to construct an optimal binary tree by considering the most probable key, Keyk, for the root, and constructing the left and right subtrees for Key1, Key2, … , Keyk-1 and Keyk+1, Keyk+2, … , Keyn recursively in the same way.

Consider the following example:

Bat Cat Dog Hog Rat

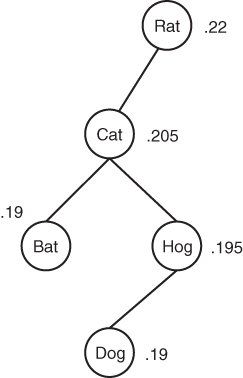
.19 .205 .19 .195 .22

Sort in descending order of probability:

Rat Cat Hog Bat Dog

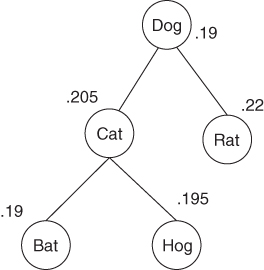
.22 .205 .195 .19 .19

Scan the sorted array from left to right and insert into an initially empty binary tree:



Expected symbol length = .22x1 + .205x2 + .19x3 + .195x3 + .19x4 = 2.545

Complexity analysis: while sorting is done in θ(n·lgn), in the worst case (when the keys are already sorted), the tree is a chain, and the insertions require 1 + 2 + … + (n-1) = n(n-1) node accesses → θ(n2)



Note that this tree is not optimal; a better one is

Expected search = .19 + .85 + .155 = 2.195

**37)** Suppose we assign n persons to n jobs. Let Cij be the cost of assigning the ith person to the jth job. Use a greedy approach to write an algorithm that ﬁnds an assignment that minimizes the total cost of assigning all n persons to all n jobs. Analyze your algorithm and show the results using order notation.

For illustration, consider the following matrix of costs:

|  |  |  |  |
| --- | --- | --- | --- |
|  | J1 | J2 | J3 |
| P1 | 10 | 5 | 5 |
| P2 | 2 | 4 | 10 |
| P3 | 5 | 1 | 7 |

This assignment problem can be solved with the following greedy algorithm:

1. Create a set F of assigned jobs, initially empty.
2. For i from 1 to n: //for each row in the matrix
   1. Find the minimum cost Cij, where j is not in F.
   2. Assign job j to person i.
   3. Include j in the set F

In the example above, the algorithm assigns P1 → J2, P2 → J1, P3 → J3, with a total cost of 14.

Note that this is not in general the optimum, e.g. assigns P1 → J3, P2 → J1, P3 → J2, has a total cost of only 8.

Complexity analysis: the loop 2 executes n times, each iteration involving finding the minimum of at most n elements, which can be done in n comparisons, yielding (n2).

**38)** Use the dynamic programming approach to write an algorithm for the problem of Exercise 37. Analyze your algorithm and show the results using order notation.

For illustration, consider the following matrix of costs:

|  |  |  |  |
| --- | --- | --- | --- |
|  | J1 | J2 | J3 |
| P1 | 10 | 5 | 5 |
| P2 | 2 | 4 | 10 |
| P3 | 5 | 6 | 7 |

This optimal assignment problem can be solved with the so-called Hungarian Algorithm:

1. Row reduction: Find the minimum element in each row and subtract it from all the elements on that row. For our example, this results in the matrix:

|  |  |  |  |
| --- | --- | --- | --- |
|  | J1 | J2 | J3 |
| P1 | 5 | 0 | 0 |
| P2 | 0 | 2 | 8 |
| P3 | 0 | 1 | 2 |

1. Column reduction: Find the minimum element in each column and subtract it from all the elements on that column. In our example, each column already has a zero, so there are no changes.
2. Cover the minimum number of row and columns such that all zeroes are covered. In our example, this is column 1 and row 1:

|  |  |  |  |
| --- | --- | --- | --- |
|  | J1 | J2 | J3 |
| P1 | 5 | 0 | 0 |
| P2 | 0 | 2 | 8 |
| P3 | 0 | 1 | 2 |

If the number equals n, and assignment exists, if not go to step 4. In our example 2 < 3, so we go to step 4.

1. Let x be the smallest uncovered number. Subtract x from all uncovered numbers and add it to all numbers covered twice. (Do nothing to the numbers covered once.) Then go back to step 3. In our example, x = 1, and the matrix becomes:

|  |  |  |  |
| --- | --- | --- | --- |
|  | J1 | J2 | J3 |
| P1 | 6 | 0 | 0 |
| P2 | 0 | 1 | 7 |
| P3 | 0 | 0 | 1 |

The complexity of the Hungarian Algorithm: n2 for step 1, n2 for step 2, and the loop at steps 3-4 can execute for a maximum of n-1 times, each time having to find and subtract the minimum of at most (n-1)2 elements. The loop dominates, yielding (n3).

Note: For clarity, the details about how to find the covering lines in step 3 were omitted. The procedure is somewhat technical, involving a sequence of “stars” and “primes”, as described below. Since only (n2) operations are required, the result of (n3) still stands .

A. Find a non-covered zero and prime it. If there is no starred zero in the row containing this primed zero, go to step 6. Otherwise, cover this row and uncover the column containing the starred zero. Continue in this manner until there are no uncovered zeros left. Save the smallest uncovered value and go to step 4.

B. Construct a series of alternating primed and starred zeros as follows. Let Z0 represent the uncovered primed zero found at A. Let Z1 denote the starred zero in the column of Z0 (if any). Let Z2 denote the primed zero in the row of Z1 (there will always be one). Continue until the series terminates at a primed zero that has no starred zero in its column. Unstar each starred zero of the series, star each primed zero of the series, erase all primes, and uncover every line in the matrix.

**39)** Use a greedy approach to write an algorithm that minimizes the number of

record moves in the problem of merging n files. Use a two-way merge pattern.

(Two files are merged during each merge step.) Analyze your algorithm, and

show the results using order notation.

Let the files be F1, F2, …, Fn, each containing r1, r2, …, rn records, respectively.

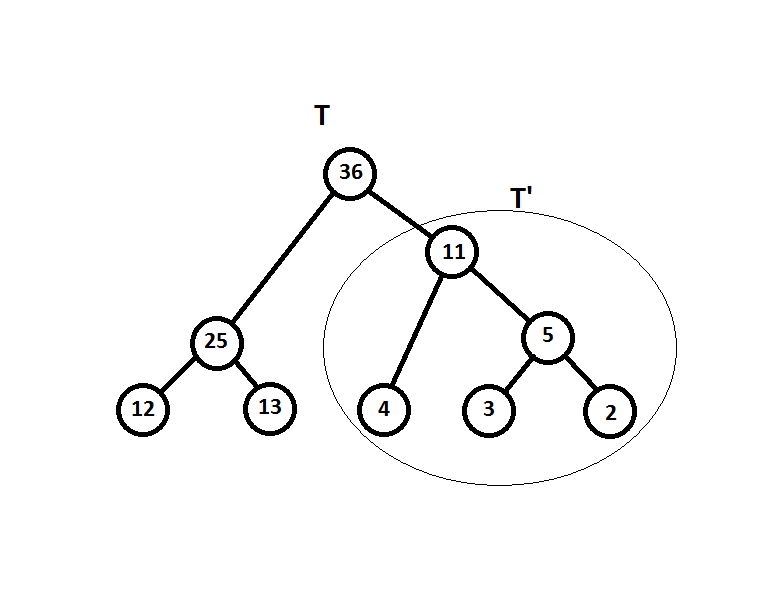
1. For i from 1 to n-1
   1. Pick the two shortest files //This is greedy!
   2. Merge them //See Algorithm 2.3 - Merge
   3. Remove the two files
   4. Insert the merged file

The exercise does not require to prove that the algorithm is optimal, but a simple proof exists.

Complexity analysis: The for loop executes (n-1) times. If we use a priority queue (see Section 4.4.2) to store the files, picking and removing the two shortest is done in constant time, and inserting the merged file in the queue is (lgn). The total is (n·lgn).

Note: The total above does not include the cost of actually merging the files, rather it describes the complexity of finding only the sequence of merges. Accordingly, the Merge step in the pseudocode above should be understood as simply adding up the numbers of records in the two files.

**40)** Use the dynamic programming approach to write an algorithm for Exercise 39. Analyze your algorithm and show the results using order notation.

The merging process is best visualized as a tree having the initial files as leaves. Every two-way merge creates a parent node whose number of records is the sum of the two children:

In order to apply dynamic programming, we first make sure that the problem has optimal sub-structure. Indeed, if we consider any subtree T' of an optimal tree T, the merging order in T' must be optimal, because otherwise we could replace T' with a T'' of smaller cost, which would result in a smaller cost of T!

Let the files be F1, F2, …, Fn (random order), each containing r1, r2, …, rn records, respectively.

We construct partial solutions bottom-up, and place the results in an (n-1)x(n-1) matrix M. Element M[k,i] contains the minimum cost of merging k files, where the left subtree is a merge of i files (and the right is a merge of k-i files):

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 2 files → | 1,1 |  |  |  |  |  |  |
| 3 files → | 1,2 | 2,1 |  |  |  |  |  |
| 4 files → | 1,3 | 2,2 | 3,1 |  |  |  |  |
| …. |  |  |  |  |  |  |  |
| k files → | 1, k-1 | 2, k-2 | 3, k-3 | …. | **i, k-i** | …. |  |
| …. |  |  |  |  |  |  |  |
| n files → | 1, n-1 | 2, n-2 | 3, n-3 | …. | i, n-i | … | n-1, 1 |

k files can be chosen in ways among the total n, so the third dimension of the matrix consists of all the possible such subsets. The general element M[k,i] has therefore a third index, denoting the particular set of k chosen; let’s call this index S (uppercase). Finally, a subset of i out of the k can in turn be chosen in ways, and let’s call this index s (lowercase) → M[k,S,i,s], k = 2…n, i = 1,k-1.

The recursion formula is

**M[k,S,i,s] = min[M(i,Si,j,sj) + M(k-i,Sk-i,t,st)]** ,

taken over all possible partitions of S into Si, Sk-i, and all pairs sj, st.

We calculate the number of sums over which the min above is taken: For each of the subsets Si of S, the sub-subset sj can be chosen in ways, and the sub-subset st in ways, so there are

 terms. Making use of a well-known combinatorial identity (Example A.10), we get (2i-2)(2k-i-2) ≈ 2k terms. Since finding the maximum is (n), we have the same number of comparisons for one entry in “row” k.

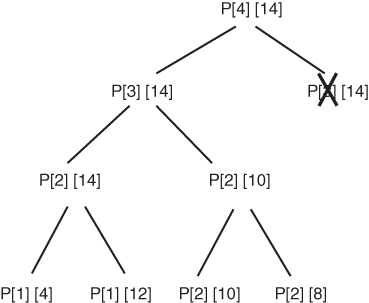
For the entire “slice” k, we have about 2k , which as above is ≈ 2k2k-1 = 22k/2 = 4k/2 comparisons.

Finally, for the entire table we have 4k = =  , using the geometric series identity (Example A.4). So the divide-and-conquer algorithm described is **(4n)**.

**41)** Prove that the greedy approach to the Fractional Knapsack problem yields an optimal solution.

Proof by contradiction: Assume there is an optimal solution S’ that does not coincide with the solution S obtained by the algorithm in Section 4.5.2. Let ***itemi*** be the first item in S where the corresponding item in S’ is different. Let ***itemk***  be the item that has replaced it in S’. Because the items are sorted by density, k > i, so it is possible to replace the quantity of ***itemk*** in S’ by the corresponding quantity of ***itemi*** which has not been used yet in S’. (If there is not enough of ***itemi***, replace only wi.) With this modification, the value of S’ has increased, which contradicts the optimality of S’. Q.e.d.

**42)** Show that the worst-case number of entries computed by the refined dynamic programming algorithm for the 0-1 Knapsack problem is in (2n). Do this by considering the instance in which W = 2n – 2 and wi = 2i–1 for 1 ≤ i ≤ n.

Consider first an example:

wi = [0 2 4 8] 1 ≤ i ≤ 4

w = 14 if n = 4

The total number of array entries computed is

1 + 1 + 2 + 22

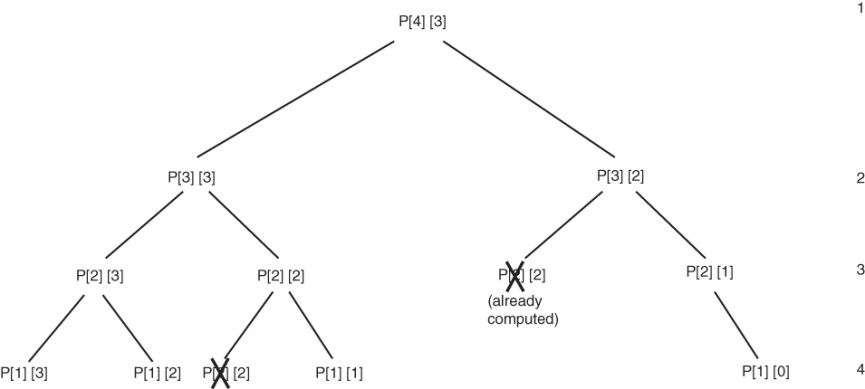
Generalizing for n we have

1 + (1 + 2 + 22 +⋅…⋅+2n) = 1 + |(2n + 1 – 1)

=2n + 1 = 2·2n = (2n)

**43)** Show that in the refined dynamic programming algorithm for the 0-1 Knapsack problem, the total number of entries computed is about (W + 1) x (n + 1) / 2, when n = W + 1 and wi = 1 for all i.

Let’s examine first the example n = 4, W = 3, w = [1 1 1 1], 1 ≤ i ≤ 4

(W + 1) x (n + 1) / 2 = 10, which exactly matches the number of entries:

Generalizing, the total number of array entries computed is



When n = w + 1, the total array entries computed is therefore .

**Additional Exercises**

**44)** Show with a counterexample that the greedy approach does not always yield an optimal solution for the Change problem when the coins are U.S. coins and we do not have at least one of each type of coin.

If we do not have any nickels, but have any number of quarters, dimes, and pennies, and we use greedy approach to find change $0.40, the greedy approach gives

.25 + .10 + .1 + .1 + .1 + .1 + .1 = .40, using 7 coins.

However, we could non-greedily find .10 + .10 + .10 + .10 = .40, which uses only 4 coins.

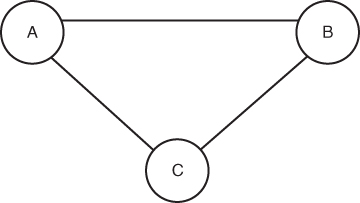
**45.** Prove that a complete graph (a graph in which there is an edge between every pair of vertices) has nn−2 spanning trees. Here n is the number of vertices in the graph.

Solution: This result was first found by Carl Wilhelm Borchardt in 1860, so here we shall denote the number by B(n). While there are many proofs (and generalizations), the most intuitve is probably Jim Pitman’s (as explained in Aigner, Ziegler’s “Proofs from THE BOOK”, Springer-Verlag, 1998):

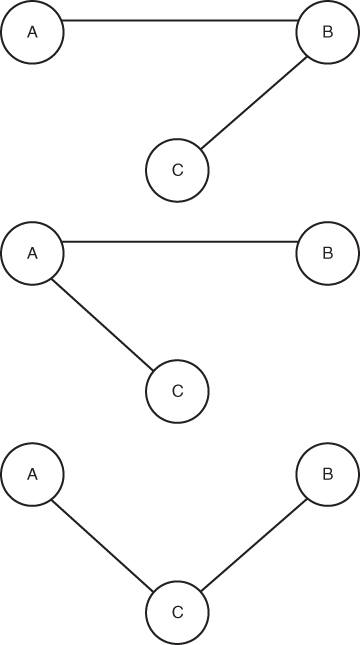
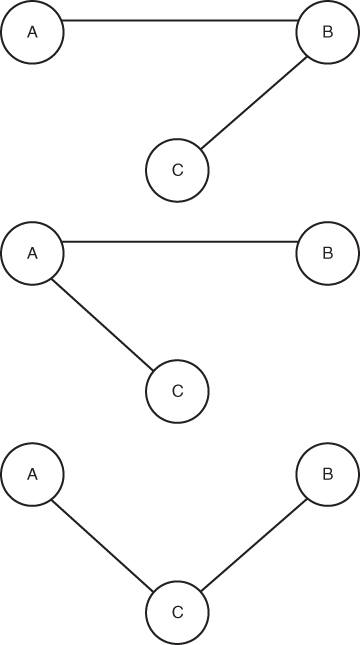
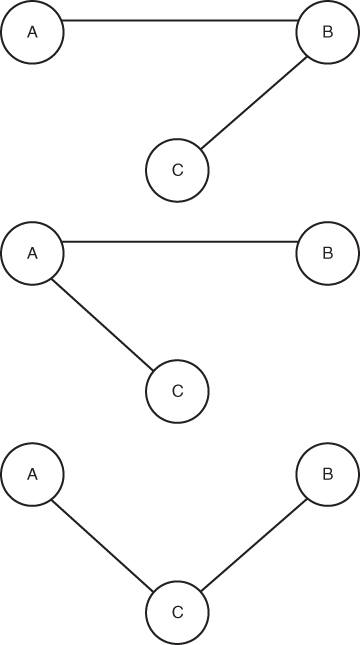
In the complete graph with n nodes, we count the ways in which we can add edges to build a spanning tree, starting from an arbitrary node. We can do this in two ways:

1. First we choose any of the B(n) spanning trees. Then we can pick the root of the tree in n possible ways. For each of the roots, we can add the (n-1) edges of the tree in (n-1)! Ways, for a total of n(n-1)!B(n) = n!B(n) ways.
2. We put together he tree by picking random edges at each step. The first edge can start in any of the n vertices of the graph, and it can end in any of the remaining (n-1) vertices, so we have n(n-1). The second edge can also start in any of the n vertices, but it can only end in the remaining (n-2), etc. Since we have (n-1) edges to add, the product is nn-1(n-1)∙(n-2)∙…∙1 = n!nn-2.

Comparing the results from A and B above, we can simplify n! to obtain B(n) = nn-2.

As an example, consider this complete graph with n = 3:

It should have *nn* – 2 spanning tree = 33 – 2 = 3. Indeed, below are shown all the possible spanning trees, and there are 3 of them:

**46)** Use a greedy approach to write an algorithm for the traveling salesman problem. Show that your algorithm does not always find a minimum-length tour.

Nearest-neighbor algorithm:

Initialize:

set of unused vertices is all vertices in graph: U = V

list of nodes in the Hamiltonian circuit is empty: H = []

Start with a randomly chosen vertex vi → add vi to H

Repeat (n-1) times:

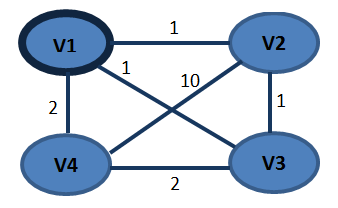
Choose the edge of minimum weight that starts in vi: e = (vi, vj)

Add vj to H

Remove vi from U

This is obviously Q(n2) in the worst case, since at any of the (n-1) vertices we may have to examine (n-1) edges. In a general graph, it can fail to find a circuit, even if such a circuit exists, but then the problem of finding a circuit is NP-complete anyway (see ch.9), so it cannot be solved greedily.

The Nearest-neighbor algorithm is, however, guaranteed to find a cycle in a **complete graph**, so this is what we shall use for the example:



Here the optimal cycle starting at v1 is obviously v1 → v2 → v3 → v4, of cost 6, but the algorithm finds v1 → v3 → v2 → v4, of cost 14.

**47)** Prove that the algorithm for the Multiple-Server Scheduling problem of Exercise 19 always ﬁnds an optimal schedule.

As described at the end of Section 4.3.1, the schedule S is:

Server 1 serves jobs 1, (1+m), (1+2m), (1+3m),...

Server 2 serves jobs 2, (2+m), (2+2m), (2+3m),...

…

Server i serves jobs i, (i+m), (i+2m), (i+3m),..., (i+km), …

…

Server m serves jobs m, (m+m), (m+2m),(m+3m),...

Assume by contradiction that S is not optimal, and let S’ be an optimal schedule. S and S’ are different, so let (i,k) be the first matrix index where they differ, i.e. instead of job i+km, server i has in S’ another job, whose index in S is j+lm, and Time(i+km) < Time(j+lm). If we swap in S’ the jobs i+km and j+lm, we can shift left all jobs scheduled after that. Let’s call the resulting schedule S’’.

On server i, the total time for S’’ is shorter. On the other hand, the server where j+lm went still has the same jobs before the end of j+lm, so that end (and everything after it) is still scheduled at the same time. This shows that the net change is that Time(S’’) < Time(S’), which contradicts the optimality of S’.

**48)** Without constructing a Huﬀman tree, generate Huﬀman code for a given set of characters.

At each step, we merge the two symbols of minimal probability into a new symbol with probability equal to the sum, and re-sort the list of symbols.

For example, let *A* = {*a*1, *a*2, *a*3, *a*4, *a*5} and *P* (*ai*) = {0.2, 0.4, 0.2, 0.1, 0.1}

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Symbol | Step 1 | Step 2 | Step 3 | Step 4 | | Code word | Reverse code |
| *a*2 | 0.4 | 0.4 | 0.4 | 0.6 | 0 | 1 | 1 |
| *a*1 | 0.2 | 0.2 | 0.4 0 | 0.4 | 1 | 10 | 01 |
| *a*3 | 0.2 | 0.2 0 | 0.2 1 |  | | 000 | 000 |
| *a*4 | 0.1 0 | 0.2 1 |  |  | | 0100 | 0010 |
| *a*5 | 0.1 1 |  |  |  | | 1100 | 0011 |

**49)** Generalize Huﬀman’s algorithm to ternary codewords and prove that it yields optimal ternary codes.

Ternary Huffman tree:

1.Sort the symbols by probability.

2.If nr. of symbols is odd, add a “filler” symbol with probability 0. This way we always have an even nr. of symbols from now on.

2. Combine the three symbols with the lowest 3 probabilities.

3. Assign the sum of the 3 probabilities to the combined symbol.

4. Go back to 1.

Example:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *X* = | *x*1 | *x*2 | *x*3 | *x*4 | *x*5 | *x*6 | *x*7 |
|  | 0.50 | 0.26 | 0.11 | 0.04 | 0.04 | 0.03 | 0.02 |
|  |  |  |  |  |  |  |  |

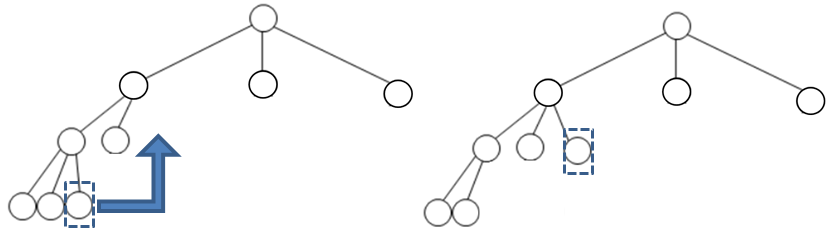
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  |  |  | codewords |
| *x*1 | 0.50 | 0.50 | 0.5 | 1 | 0 |
| *x*2 | 0.26 | 0.26 | 0.26 |  | 1 |
| *x*3 | 0.11 | 0.11 | 0.24 |  | 20 |
| *x*4 | 0.04 | 0.04 |  |  | 21 |
| *x*5 | 0.04 | 0.9 |  |  | 220 |
| *x*6 | 0.03 |  |  |  | 221 |
| *x*7 | 0.02 |  |  |  | 222 |

The average nr. of trits for this code is 0.50 x 1 + 0.50 x 2 = 1.33 trit/symbol.

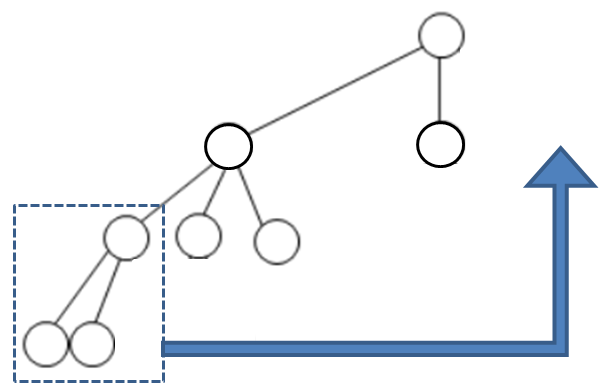
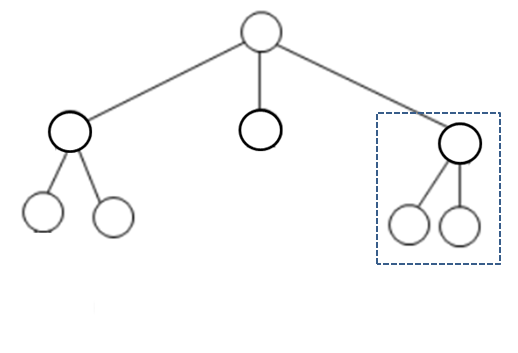
The proof closely parallels the one in the text for the binary Huffman code:

First we have the equivalent of Lemma 4.4 (solved in Exercise 32), which says that an optimal ternary tree is always full. This is proved with the same type of argument: if a node has fewer than 3 children, there are 2 possibilities:

* 1 child → As in Exercise 32, we move up the entire subtree rooted at the child, thus obtaining a tree of smaller cost.
* 2 children → We move up one of the grandchildren, thus obtaining a tree of smaller cost, as illustrated below:



If there are no grandchildren, then, because the total nr. of nodes is odd, somewhere higher in the tree must be another node with only 2 children. If we move the node in the higher position, we have again a tree of smaller cost:

This proves the lemma.

Now we have the equivalent of Theorem 4.5: The ternary Huffman algorithm produces an optimal tree.

The proof is by induction, just like the proof in the text.

**50)** If the characters are sorted according to their frequencies, then the priority queue used in Huffman’s algorithm on page 180 can be replaced with two standard queues to keep tracking of the two highest priority elements. Initially, the first queue contains all of the characters in their frequency-sorted order. At each iteration of the loop, the front of both queues is examined and the elements with the lowest frequencies are removed from the head of either one or both queues. When two nodes are combined, they are inserted into the rear of the second queue with a frequency that is the sum of their individual frequencies. The process is repeated n − 1 times as in the original algorithm.

Since each call to the remove or insert function of either queue takes constant time, instead of log time, each iteration of the loop executes seven constant-time operations, and thus the entire loop executes 7\*(n − 1) constant-time operations. The algorithm is thus O(n).