Solutions to Homework 7

22C:044 Algorithms, Fall 2000

1. The $p$'th slot to try after the initial position $h(k)$ is

$$h(k) + 1 + 2 + \ldots + p \equiv h(k) + p(p+1)/2 \equiv h(k) + p^2/2 + p/2 \pmod{m}.$$  

So the proposed scheme is quadratic probing with $c_1 = c_2 = 1/2$.

Let $m = 2^t$ be a power of two. The $p$'th and the $q$'th probe positions of key $k$ are $h(k) + p^2/2 + p/2$ and $h(k) + q^2/2 + q/2$, respectively. Let us show that these two positions are distinct for any $p \neq q$, $0 \leq p, q < 2^t$. Namely, if

$$h(k) + p^2/2 + p/2 \equiv h(k) + q^2/2 + q/2 \pmod{2^t}$$

then

$$(p - q)(p + q + 1)/2 \equiv 0 \pmod{2^t}.$$ 

This means that $(p - q)(p + q + 1)$ is divisible by $2^{t+1}$. Because $p - q$ or $p + q + 1$ is odd, either $p - q$ or $p + q + 1$ must be divisible by $2^{t+1}$. Number $p + q + 1$ can not be a multiple of $2^{t+1}$ because

$$p + q + 1 \leq (2^t - 1) + (2^t - 1) + 1 = 2^{t+1} - 1.$$ 

But neither can $p - q$ be a multiple of $2^{t+1}$ because $p \neq q$.

We conclude that the probe positions with $p = 0, 1, 2, \ldots, 2^t - 1$ are all distinct, so all $2^t$ positions are among them.

2. Knapsack($s[1\ldots n]$, S)
   1. allocate array $c[0\ldots S]$
   2. for $i \leftarrow 1$ to S do $c[i]$=nil
   3. $c[0]$=0
   4. for $i \leftarrow 1$ to n do
   5.   for $j \leftarrow s[i]$ to S do
   6.     if $(c[j]$=nil) and $(c[j-s[i]] \neq$ nil) then $c[j]$=i
   7. if $c[S]$=nil then return FALSE
   8. $j \leftarrow S$
   9. while $(j > 0)$ do print $c[j]$; $j \leftarrow j-s[c[j]]$

First we build an array $c[0\ldots S]$ whose element $c[j]$ is the smallest number $i$ such that some subset of the first $i$ rods has total length exactly $j$. If there is no subset whose sum is $i$ then we set $c[i]$=nil. The array is built on lines 1–6 of the pseudo-code. On line 6 we test whether length $j$ can be made of rod number $i$ and some subset of the first $i - 1$ rods.
If $c[S]$ remains $\textbf{nil}$ then no subset has total length $S$ and we return value $\text{FALSE}$ (line 7). Otherwise, we print out the rods whose lengths sum up to $S$ (lines 8–9).

The time complexity is dominated by the two nested $\textbf{for}$ -loops on lines 4–5. The total time complexity is $\Theta(nS)$. The space complexity is $\Theta(S)$.

3 Assumptions about the input: $A[0]$ and $A[n]$ are the beginning and the end positions of the string, $A[1..n-1]$ are the positions of the break points that need to be made, ordered from left to right.

**Breaks($A[0...n]$)**
1. allocate array $c[0...n][0...n]$
2. for $k$ ← 1 to $n$
3. for $a$ ← 0 to $n-k$
4. begin
5. $b$ ← $a+k$
6. if ($k=1$) then $c[a][b]$ ← 0 else
7. begin
8. $\text{min}$ ← $\infty$
9. for $x$ ← $a+1$ to $b-1$
10. begin
11. $w$ ← $c[a][x]+c[x][b]$
12. if $w<\text{min}$ then $\text{min} ← w$
13. end
14. $c[a][b]$ ← $\text{min}+(A[b]-A[a])$
15. end
16. end
17. PrintBreaks($A[0...n]$)

**PrintBreaks($A[a...b]$)**
1. if ($b-a < 2$) then return else
2. for $x$ ← $a+1$ to $b-1$
3. if $c[a][b] = c[a][x]+c[x][b] +(A[b]-A[a])$ then
4. begin
5. Print $x$
6. PrintBreaks($a,x$)
7. PrintBreaks($x,b$)
8. return
9. end
The algorithm resembles the optimal triangulation algorithm. Element \( c[a][b] \) will store the minimum cost of the cuts \( A[a+1 \ldots b-1] \) between positions \( A[a] \) and \( A[b] \), assuming that cuts has been made at positions \( A[a] \) and \( A[b] \). We try all possible choices \( x \) for the next cut to be made between positions \( A[a] \) and \( A[b] \), and choose one that gives the smallest total cost. Value \( (A[b]-A[a]) \) is the cost of the cut \( x \), and \( c[a][x] \) and \( c[x][b] \) are the costs of making the remaining cuts between positions \( A[a] \) and \( A[x] \), and \( A[x] \) and \( A[b] \), respectively.

In the end, a recursive **PrintBreaks** is used to print the cuts in the optimal order.

4 Let us assume the points are given in the order from left to right. In other words, point 1 is the leftmost point and point \( n \) is the rightmost point. Let \( w_{i,j} \) be the distance between points \( i \) and \( j \).

A bitonic tour through the points consists of a left-to-right path from point 1 to point \( n \), followed by a right-to-left path from \( n \) back to 1. All points must be visited.

Clearly points \( n \) and \( n-1 \) are connected on every bitonic tour. Let us define a bitonic path from node \( i \) to node \( i-1 \) as follows: it is a path that starts with a right-to-left path from point \( i \) to point 1, followed by a left-to-right path from point 1 to point \( i-1 \). All points 1, 2, 3, \ldots \( i \) must be on the path.

Let \( P \) be such a bitonic path from node \( i \) to node \( i-1 \). Let \( j \) be the second node on the path, that is, the node connected to \( i \). Then nodes \( j+1, j+2, \ldots, i-1 \) must be the last nodes of path \( P \). This means that \( P \) consists of edge \( i \rightarrow j \), a bitonic path between points \( j+1 \) and \( j \), and edges \( j+1 \rightarrow j+2 \rightarrow \ldots \rightarrow i-1 \). If we know the lengths of the shortest bitonic paths between points \( j+1 \) and \( j \) for all \( j < i-1 \) then we can easily find the length of the shortest bitonic path between nodes \( i \) and \( i-1 \) by trying all choices of \( j \) and choosing the one that gives the shortest path.

We use an array \( c[2 \ldots n] \) whose element \( c[i] \) stores the length of the shortest bitonic path between nodes \( i \) and \( i-1 \). The shortest bitonic tour must then have length \( c[n] + w_{n,n-1} \).
Here’s the first part of the algorithm. Variable \texttt{sum} is used to accumulate the sum of the lengths of the path \( j+1 \rightarrow j+2 \rightarrow \ldots \rightarrow i-1 \).

Bitonic
1. allocate array \( c[2...n] \)
2. \( c[2] \leftarrow w_{1,2} \)
3. for \( i \leftarrow 3 \) to \( n \) do
4. begin
5. \( \text{min} \leftarrow \infty \)
6. \( \text{sum} \leftarrow 0 \)
7. for \( j \leftarrow i-2 \) downto 1 do
8. begin
9. \( w \leftarrow c[j+1] + w_{j,i} + \text{sum} \)
10. if \( w < \text{min} \) then \( \text{min} \leftarrow w \)
11. \( \text{sum} \leftarrow \text{sum} + w_{j,j+1} \)
12. end
13. \( c[i] \leftarrow \text{min} \)
14. end

Once the array \( c[2...n] \) is filled we can make a second pass to print out the actual line segments:

15. print line segment \( n \leftarrow n-1 \)
16. \( i \leftarrow n \)
17. while \( i > 2 \) do
18. begin
19. \( \text{min} \leftarrow \infty \)
20. \( \text{sum} \leftarrow 0 \)
21. for \( j \leftarrow i-2 \) downto 1 do
22. if \( c[i] \neq c[j+1] + w_{j,i} + \text{sum} \) then \( \text{sum} \leftarrow \text{sum} + w_{j,j+1} \) else
23. begin
24. print line segment \( i \leftrightarrow j \)
25. for \( k \leftarrow j+1 \) to \( i-2 \) do
26. print line segment \( k \leftrightarrow k+1 \)
27. \( i \leftarrow j+1 \)
28. break (=goto 17)
29. end
30. end
31. print line segment \( 1 \leftarrow 2 \)