Analysis of open-address hashing

• Let us express our analysis of in terms of the load factor $\alpha = n/m$ of the hash table
• Now at most one element occupies each slot, and thus $n \leq m$, which implies $\alpha \leq 1$
• Assume that we are using uniform hashing
• In this idealized scheme, the probe sequence \( \langle h(k, 0), h(k, 1), \ldots, h(k, m - 1) \rangle \) used to insert or search for each key \( k \) is equally likely to be any permutation of \( \langle 0, 1, \ldots, m - 1 \rangle \)

Theorem 11.6 Given an open-address hash table with $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1 - \alpha)$, assuming UH.

Proof Every probe but the last accesses an occupied slot that does not contain the desired key, and the last slot probed is empty.

Define the random variable \( X \) to be the number of probes made in an unsuccessful search, and also define the event \( A_i, i = 1, 2, \ldots \), to be the event that an \( i \)th probe occurs and it is to an occupied slot.
The event \( \{ X \geq i \} \) is the intersection of events \( A_1 \cap A_2 \cap \cdots \cap A_{i-1} \).

Bound \( \Pr\{X \geq i\} \) by \( \Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\} \) which by Exercise C.2-5
\[
\Pr\{A_{i-1} | A_1 \cap A_2 \cap \cdots \cap A_{i-2}\} = \Pr\{A_1\} \cdot \Pr\{A_2 | A_1\} \cdot \Pr\{A_3 | A_1 \cap A_2\} \cdots
\]

There are \( n \) elements and \( m \) slots, so
\( \Pr\{A_1\} = \frac{n}{m} \).

For \( j > 1 \), the probability that there is a \( j \)th probe and it is to an occupied slot, given that the first \( j-1 \) probes were to occupied slots, is
\[
(n - j + 1) / (m - j + 1)
\]

This probability follows because we would be finding one of the remaining \((n - j + 1)\) elements in one of the \((m - j + 1)\) unexamined slots, and by the assumption of UH, the probability is the ratio of these quantities.

\( n < m \) implies that \((n - j) / (m - j) \leq n / m \) for all \( 0 \leq j < m \).

Therefore, we have for all \( 1 \leq i \leq m \),
\[
\Pr\{X \geq i\} = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdots \frac{n-i+2}{m-i+2} \leq \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}
\]
• Now, because $E[X] = \sum_{i=1}^{\infty} \Pr\{X \geq i\}$

\[
E[X] = \sum_{i=1}^{\infty} \Pr\{X \geq i\} \\
\leq \sum_{i=1}^{\infty} \alpha^{i-1} \\
= \sum_{i=0}^{\infty} \alpha^{i} \\
= \frac{1}{1 - \alpha}
\]

This bound of \( \frac{1}{1 - \alpha} = 1 + \alpha + \alpha^2 + \cdots \) has an intuitive interpretation

– We always make the first probe
– With probability approximately \( \alpha \), it finds an occupied slot, so that we need to probe again
– With probability approx. \( \alpha^2 \), the first two slots are occupied and we make a third probe, …

• If \( \alpha \) is a constant, Theorem 11.6 predicts that an unsuccessful search runs in \( O(1) \) time
• If the table is half full, the avg. number of probes in an unsuccessful search is \( \leq \frac{1}{1 - .5} = 2 \)
• If it is 90% full, the average number of probes is \( \leq \frac{1}{1 - .9} = 10 \)
Corollary 11.7  Inserting an element into an open-address hash table with load factor $\alpha$ requires at most $\frac{1}{1 - \alpha}$ probes on average, assuming uniform hashing.

Proof  An element is inserted only if there is room in the table, and thus $\alpha < 1$. Inserting a key requires an unsuccessful search followed by placing the key into the first empty slot found. Thus, the expected number of probes is at most $\frac{1}{1 - \alpha}$.

Theorem 11.8 Given an open-address hash table with load factor $\alpha < 1$, the expected number of probes in a successful search is at most

$$\frac{1}{\alpha \ln \frac{1}{1 - \alpha}}$$

assuming UH and assuming that each key in the table is equally likely to be searched for.

- If the hash table is half full, the expected number of probes in a successful search is $< 1.387$
- If the hash table is 90 percent full, the expected number of probes is $< 2.559$
14 Augmenting Data Structures

- Some engineering situations require more than a “textbook” data structure
- Usually it suffices to augment a textbook data structure by storing additional information in it
- You can then program new operations for the data structure to support the desired application
- The added information must be updated and maintained by the ordinary operations on the data structure

14.1 Dynamic order statistics

- Let us see how to modify red-black trees (RBTs) so that we can determine any order statistic for a dynamic set in \(O(\lg n)\) time
- We shall also see how to compute the rank of an element—its position in the linear order of the set—in \(O(\lg n)\) time
- An order-statistic tree \(T\) is simply a red-black tree with additional information stored in each node
Besides the usual RBT attributes $x.key$, $x.color$, $x.p$, $x.left$, and $x.right$ in a node $x$, we have another attribute, $x.size$

This attribute contains the number of (internal) nodes in the subtree rooted at $x$ (including $x$ itself), that is, the size of the subtree.

If we define the sentinel's size to be 0—that is, we set $T.nil.size$ to be 0—then we have the identity

$$x.size = x.left.size + x.right.size + 1$$
We do not require keys to be distinct in an order-statistic tree.

In the presence of equal keys, the above notion of rank is not well defined.

We remove this ambiguity for an order-statistic tree by defining the rank of an element as the position at which it would be printed in an inorder walk of the tree.

In previous figure, e.g., the key 14 stored in a black node has rank 5, and the key 14 stored in a red node has rank 6.

Retrieving an element with a given rank

Let us begin with an operation that retrieves an element with a given rank.

Procedure OS-SELECT\((x, i)\) returns a pointer to the node containing the \(i\)th smallest key in the subtree rooted at \(x\).

To find the node with the \(i\)th smallest key in an order-statistic tree \(T\), we call

\[\text{OS-SELECT}(T.\text{root}, i)\]
OS-SELECT\((x, i)\)
1. \(r \leftarrow x.\text{left.size} + 1\)
2. if \(i = r\)
3. return \(x\)
4. elseif \(i < r\)
5. return OS-SELECT\((x.\text{left}, i)\)
6. else return OS-SELECT\((x.\text{right}, i - r)\)

Consider a search for the 17th smallest element in the order-statistic tree of previous figure
- First, \(x\) is the root, whose key is 26, and \(i = 17\)
- The size of 26’s left subtree is 12, its rank is 13
- Thus, the node with rank 17 is the \(17 - 13 = 4\)th smallest element in 26’s right subtree
- Now, \(x\) becomes the node with key 41, and \(i = 4\)
- The size of 41’s left subtree is 5, its rank within its subtree is 6
- Thus, the node with rank 4 is the 4th smallest element in 41’s left subtree
• After the recursive call, \( x \) is the node with key 30, and its rank within its subtree is 2

• Thus, we recurse once again to find the \( 4 - 2 = 2 \)nd smallest element in the subtree rooted at the node with key 38

• We now find that its left subtree has size 1, which means it is the second smallest element

• Thus, the procedure returns a pointer to the node with key 38

• Because each recursive call goes down one level in the order-statistic tree, the total time for OS-SELECT is at worst proportional to the height of the tree

• Since the tree is a red-black tree, its height is \( O(\lg n) \), where \( n \) is the number of nodes

• Thus, the running time of OS-SELECT is \( O(\lg n) \) for a dynamic set of \( n \) elements
Determining the rank of an element

- OS-RANK returns the position of \( x \) in the linear order determined by an inorder tree walk of \( T \)

OS-RANK(\( T, x \))

1. \( r \leftarrow x. left. size + 1 \)
2. \( y \leftarrow x \)
3. while \( y \neq T. root \)
4. if \( y = y. p. right \)
5. \( r \leftarrow r + y. p. left. size + 1 \)
6. \( y \leftarrow y. p \)
7. return \( r \)

- E.g., when we run OS-RANK to find the rank of key 38, we get the following sequence of values of at the top of the while loop:

<table>
<thead>
<tr>
<th>iteration</th>
<th>y.key</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>38</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>30</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>41</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>26</td>
<td>17</td>
</tr>
</tbody>
</table>

- The procedure returns the rank 17
- the running time of OS-RANK is at worst proportional to the height of the tree: \( O(\lg n) \) on an \( n \)-node order-statistic tree
Maintaining subtree sizes

- Given the size attribute in each node, OS-SELECT and OS-RANK can quickly compute order-statistic information.
- We must be able to efficiently maintain these attributes within the basic modifying operations on red-black trees.
- Let us see how to maintain subtree sizes for both insertion and deletion without affecting the asymptotic running time of either operation.

Insertion into a RBT consists of two phases:
- The first goes down the tree, inserting the new node as a child of an existing one.
- The second phase goes up the tree, changing colors and performing rotations to maintain the RB properties.

To maintain the subtree sizes in the first phase, we increment $x.size$ for each node $x$ on the simple path traversed.
- The new node added gets a size of 1.
- Since there are $O(\lg n)$ nodes on the traversed path, the additional cost of maintaining the size attributes is $O(\lg n)$. 
In the second phase, the only structural changes to the underlying RBT are caused by rotations, of which there are at most two.

Moreover, a rotation is a local operation: only two nodes have their size attributes invalidated.

The link around which the rotation is performed is incident on these two nodes.

Referring to the code for `LEFT-ROTATE(T, x)`, we add the following lines:

13 \( y\.size \leftarrow x\.size \)

14 \( x\.size \leftarrow x\.left\.size + x\.right\.size + 1 \)

Since at most two rotations are performed during insertion into a RBT, we spend only \( O(1) \) additional time updating size attributes in the second phase.

Thus, the total time for insertion into an \( n \)-node order-statistic tree is \( O(\log n) \), which is asymptotically the same as for an ordinary RBT.
Deletion also consists of two phases:

- the first operates on the underlying search tree
- the second causes at most three rotations and otherwise performs no structural changes

The first phase either removes one node \( y \) from the tree or moves upward it within the tree.

To update the subtree sizes, we simply traverse a simple path from node \( y \) (starting from its original position) up to the root, decrementing the size attribute of each node on the path.

Since this path has length \( O(\lg n) \) in an \( n \)-node red-black tree, the additional time spent maintaining size attributes in the first phase is \( O(\lg n) \).

We handle the \( O(1) \) rotations in the second phase of deletion in the same manner as for insertion.

Thus, both insertion and deletion, including maintaining the size attributes, take \( O(\lg n) \) time for an \( n \)-node order-statistic tree.
14.3 Interval trees

- We augment RBTs to support operations on dynamic sets of intervals
- A **closed interval** is an ordered pair of real numbers \([t_1, t_2]\), with \(t_1 \leq t_2\)
- Interval \([t_1, t_2]\) represents the set \(\{t \in \mathbb{R} : t_1 \leq t \leq t_2\}\)
- **Open** and **half-open** intervals omit both or one of the endpoints from the set, respectively
- Extending the following results to open and half-open intervals is conceptually straightforward

Intervals are convenient for representing events that each occupy a continuous period of time

- We might, e.g., wish to query a database of time intervals to find out what events occurred during a given interval
- The data structure to follow provides an efficient means for maintaining such an interval database
- We can represent an interval \([t_1, t_2]\) as an object \(i\), with attributes \(i.\text{low} = t_1\) (the low endpoint) and \(i.\text{high} = t_2\) (the high endpoint)
- We say that intervals \(i\) and \(i'\) overlap if \(i \cap i' \neq \emptyset\)
  i.e., if \(i.\text{low} \leq i'.\text{high}\) and \(i'.\text{low} \leq i.\text{high}\)
The **interval trichotomy**: exactly one of the following three properties holds:

* a) \( i \) and \( i' \) overlap,
* b) \( i \) is to the left of \( i' \) (i.e., \( i.\text{high} < i'.\text{low} \)),
* c) \( i \) is to the right of \( i' \) (i.e., \( i'.\text{high} < i.\text{low} \))

• An interval tree is a RBT that maintains a dynamic set of elements, with each element \( x \) containing an interval \( x.\text{int} \)

\( \text{INTERVAL-INSERT}(T, x) \) adds the element \( x \), whose \( \text{int} \) attribute is assumed to contain an interval, to the interval tree \( T \)

\( \text{INTERVAL-DELETE} (T, x) \) removes the element \( x \) from \( T \)

\( \text{INTERVAL-SEARCH} (T, i) \) returns a pointer to an element \( x \) in \( T \) such that \( x.\text{int} \) overlaps interval \( i \), or a pointer to the sentinel \( T.\text{nil} \) if no such element is in the set
**INTERVAL-SEARCH** \((T, i)\)

1. \(x \leftarrow T.\text{root}\)
2. while \(x \neq T.\text{nil} \text{ and } i\) does not overlap \(x.\text{int}\)
3. if \(x.\text{left} \neq T.\text{nil} \text{ and } x.\text{left}.\max \geq i.\text{low}\)
4. \(x \leftarrow x.\text{left}\)
5. else \(x \leftarrow x.\text{right}\)
6. return \(x\)

- Each iteration of the basic loop takes \(O(1)\) time
- The height of an \(n\)-node RBT is \(O(\lg n)\)
- **INTERVAL-SEARCH** procedure takes \(O(\lg n)\) time
INTERVAL-SEARCH on the previous interval tree:

- We wish to find an interval that overlaps the interval \( i = [22,25] \)
- We begin with \( x \) as the root, which contains \([16,21]\) and does not overlap \( i \)
- \( x\.left\.max = 23 \geq i\.low = 22 \) and the loop continues with \( x \) as the left child—the node containing \([8,9]\), which does not overlap \( i \)
- This time, \( x\.left\.max = 10 \leq i\.low = 22 \), and so the loop continues with the right child of \( x \)
- Because the interval \([15,23]\) stored in this node overlaps \( i \), the procedure returns this node

An unsuccessful search: we wish to find an interval that overlaps \( i = [11,14] \)

- Since the root’s interval \([16,21]\) does not overlap \( i \), and since \( x\.left\.max = 23 \geq i\.low = 11 \), we go left to the node containing \([8,9]\)
- Interval \([8,9]\) doesn’t overlap \( i \), and \( x\.left\.max = 10 \leq i\.low = 11 \), and so we go right
- No interval in the left subtree overlaps \( i \)
- Interval \([15,23]\) does not overlap \( i \), and its left child is \( T.nil \), so again we go right, the loop terminates, and we return the sentinel \( T.nil \)
To see why INTERVAL-SEARCH is correct, we must understand why it suffices to examine a single path from the root.

The basic idea is that at any node $x$, if $x.int$ does not overlap $i$, the search always proceeds in a safe direction:

- The search will definitely find an overlapping interval if the tree contains one.
15 Dynamic Programming

- Divide-and-conquer algorithms partition the problem into disjoint subproblems, recurse, and then combine the solutions
- Dynamic programming applies when the subproblems overlap—that is, when they share subsubproblems
- A dynamic-programming algorithm solves each subsubproblem just once and saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem

We typically apply dynamic programming to optimization problems, which can have many possible solutions
- Each solution has a value, and we wish to find a solution with the optimal (min or max) value
- We call such a solution an optimal solution, several solutions may achieve the optimal value
  1. Characterize the structure of an optimal solution
  2. Recursively define the value of an optimal solution
  3. Compute the value of an optimal solution, typically in a bottom-up fashion
  4. Construct an optimal solution from computed information
15.1 Rod cutting

- Serling Enterprises buys long steel rods and cuts them into shorter rods, which it then sells
- Each cut is free
- The management wants to know the best way to cut up the rods
- We assume that we know, for $i = 1, 2, \ldots$, the price $p_i$ in dollars that Serling charges for a rod of length $i$ inches
- Rod lengths are always an integral number

<table>
<thead>
<tr>
<th>Length $i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Price $p_i$</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>17</td>
<td>17</td>
<td>20</td>
<td>24</td>
<td>30</td>
</tr>
</tbody>
</table>

The rod-cutting problem is the following:
- Given a rod of length $n$ inches and a table of prices $p_i$ for $i = 1, 2, \ldots, n$,
- determine the maximum revenue $r_n$ obtainable by cutting up the rod and selling the pieces
- Note that if the price $p_n$ for a rod of length $n$ is large enough, an optimal solution may require no cutting at all
We can cut up a rod of length $n$ in $2^{n-1}$ different ways: we have an independent option of cutting, or not cutting, at distance $i$ inches from the left end, $i = 1, 2, ..., n - 1$

- When $n = 4$, there are $2^3 = 8$ ways to cut up the rod, including the way with no cuts at all
- Cutting a 4-inch rod into two 2-inch pieces produces optimal revenue $p_2 + p_2 = 5 + 5 = 10$

If an optimal solution cuts the rod into $k$ pieces, for some $1 \leq k \leq n$, then an optimal decomposition

$$n = i_1 + i_2 + \cdots + i_k$$

of the rod into pieces of lengths $i_1, i_2, ..., i_k$ provides maximum corresponding revenue

$$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k}$$

- We can frame the values $r_n$ for $n - 1$ in terms of optimal revenues from shorter rods:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$$
• To solve the original problem of size \( n \), we solve smaller problems of the same type.
• Once we make the first cut, we may consider the two pieces as independent instances of the rod-cutting problem.
• The overall optimal solution incorporates optimal solutions to the related two subproblems, maximizing revenue from each of those pieces.
• The rod-cutting problem exhibits optimal substructure: optimal solutions incorporate optimal solutions to related subproblems, which we may solve independently.

We view a decomposition as consisting of a first piece of length \( i \) cut off the left-hand end, and then a right-hand remainder of length \( n - i \).
• Only the remainder, and not the first piece, may be further divided.
• Decomposition of a length-\( n \) rod has a first piece followed by some decomposition of the rest.
• No cuts at all: first piece has size \( i = n \) and revenue \( p_n \) and the remainder has size 0 with corresponding revenue \( r_0 = 0 \).
• We thus obtain the following equation:

\[
r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})
\]
The following procedure implements the computation implicit in equation in a straightforward, top-down, recursive manner.

\[
\text{CUT-ROD}(p, n)
\]

1. if \( n = 0 \)
2. \( \text{return } 0 \)
3. \( q \leftarrow \infty \)
4. for \( i \leftarrow 1 \) to \( n \)
5. \( q \leftarrow \max(q, p[i] + \text{CUT-ROD}(p, n - i)) \)
6. \( \text{return } q \)

CUT-ROD takes as input array \( p[1..n] \) of prices and an integer \( n \).

If we ran CUT-ROD on a computer, we would find that once the input size becomes moderately large, our program would take a long time to run.

For \( n = 40 \), we would find that our program takes at least several minutes, and most likely more than an hour.

In fact, we would find that each time we increase \( n \) by 1, our program’s running time would approximately double.
Cut-Rod calls itself recursively over and over again with the same parameter values – it solves the same subproblems repeatedly.

Cut-Rod \((p, n)\) calls Cut-Rod \((p, n - i)\) for each \(j = 0, 1, \ldots, n - 1\).

The amount of work done, as a function of \(n\), grows explosively.

Let \(T(n)\) denote the total number of calls made to Cut-Rod when called with its second parameter equal to \(n\).

This equals the number of nodes in a subtree whose root is labeled \(n\) in the recursion tree.

The count includes the initial call at its root.

Thus, \(T(0) = 1\) and

\[
T(n) = 1 + \sum_{j=0}^{n-1} T(j)
\]

\(T(n) = 2^n\), the running time is exponential.
Using dynamic programming for optimal rod cutting

- We arrange for each subproblem to be solved only once, saving its solution
- We can just look the solution up again later
- Dynamic programming serves an example of a time-memory trade-off
- This approach runs in polynomial time when the number of distinct subproblems involved is polynomial in the input size and we can solve each such in polynomial time

In top-down approach with memoization, we write the procedure recursively modified to save the result of each subproblem.
- The procedure now first checks to see whether it has previously solved this subproblem.
  - If so, it returns the saved value, if not, the it computes the value in the usual manner

MEMO-CUT-Rod\((p,n)\)

1. let \(r[0..n]\) be a new array
2. for \(i \leftarrow 1\) to \(n\)
3. \(r[i] \leftarrow -\infty\)
4. return MEMO-CUT-Rod-Aux \((p,n,r)\)
MEMO-CUT-ROD-AUX\((p, n, r)\)

1. if \(r[n] \geq 0\)
2. return \(r[n]\)
3. if \(n = 0\)
4. \(q \leftarrow 0\)
5. else \(q \leftarrow -\infty\)
6. for \(i \leftarrow 1\) to \(n\)
7. \(q \leftarrow \max(q, p[i] + \text{MEMO-CUT-ROD-AUX}(p, n - i, r))\)
8. \(r[n] \leftarrow q\)
9. return \(q\)

The **bottom-up method** typically depends on some natural notion of the “size” of a subproblem.

- We sort the subproblems by size and solve them in size order, smallest first.
- When solving a subproblem, we have already solved all of the smaller subproblems its solution depends upon, we have saved their solutions.
- We solve each subproblem only once, and when we first see it, we have already solved all of its prerequisite subproblems.
BOTTOM-UP-CUT-ROD\((p, n)\)
1. let \(r[0..n]\) be a new array
2. \(r[0] \leftarrow 0\)
3. for \(j \leftarrow 1\) to \(n\)
   4. \(q \leftarrow -\infty\)
   5. for \(i \leftarrow 1\) to \(j\)
   6. \(q \leftarrow \max(q, p[i] + r[j-i])\)
   7. \(r[j] \leftarrow q\)
8. return \(r[n]\)

- The running time of \(\text{BOTTOM-UP-CUT-ROD}\) is \(\Theta(n^2)\), due to its doubly-nested loop structure
  - The number of iterations of its inner for loop, in lines 5–6, forms an arithmetic series
- The running time of its top-down counterpart, \(\text{MEMO-CUT-ROD}\), is also \(\Theta(n^2)\)
  - A recursive call to previously solved subproblem returns immediately, \(\text{MEMO-CUT-ROD}\) solves each subproblem just once
  - To solve a subproblem of size \(n\), the for loop of lines 6–7 iterates \(n\) times
  - The total number of iterations of this loop, over all recursive calls forms an arithmetic series
Reconstructing a solution

- The solutions to the rod-cutting problem return the value of an optimal solution, but they do not return an actual solution: a list of piece sizes.
- We can extend the dynamic-programming approach to record also a choice that led to the optimal value.
- An extended version of BOTTOM-UP-CUT-ROD computes, for each rod size $j$, not only the maximum revenue $r_j$, but also $s_j$, the optimal size of the first piece to cut off.

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
1. let $r[0..n]$ and $s[0..n]$ be new arrays
2. $r[0] \leftarrow 0$
3. for $j \leftarrow 1$ to $n$
4.    $q \leftarrow -\infty$
5.    for $i \leftarrow 1$ to $j$
6.        if $q < p[i] + r[j - i]$
7.            $q \leftarrow p[i] + r[j - i]$
8.            $s[j] \leftarrow i$
9.    $r[j] \leftarrow q$
10. return $r$ and $s$
```
• The following procedure prints out the complete list of piece sizes in an optimal decomposition of a rod of length $n$:

**PRINT-CUT-ROD-SOLUTION**($p, n$)

1. $(r, s) \leftarrow \text{EXTENDED-BOTTOM-UP-CUT-ROD}(p, n)$
2. while $n > 0$
3. print $s[n]$
4. $n \leftarrow n - s[n]$

In our example, the call **EXTENDED-BOTTOM-UP-CUT-ROD**($p, 10$) would return the following arrays:

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r[i]$</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>17</td>
<td>17</td>
<td>20</td>
<td>24</td>
<td>30</td>
</tr>
<tr>
<td>$s[i]$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>10</td>
</tr>
</tbody>
</table>

• A call to **PRINT-CUT-ROD-SOLUTION**($p, 10$) would print just 10, but a call with $n = 7$ would print the cuts 1 and 6, corresponding to the first optimal decomposition for $r_7$. 