7.0

Last week we covered Binary Search Trees and gave the basic alg. for them. Today we will finish this subject by comparing BSTs and Heaps and discuss shortly improvements to BST.

We will then study a new algorithmic technique called dynamic programming.

7.1 ~~6.9~~

One question is how to control the height of the BST.
Note that if we insert a sorted sequence one by one we will get a tree of depth $O(n)$. This is bad as our algorithms run in time $O(h)$.

The solution is to insert them in random order. One can show that if we pick a random permutation (on n distinct keys) then with high probability there will not be monotone sequences of length more than $O(\log n)$.

BST vs. Heaps

	BST	max. Heap
Max	$O(h)$	$O(1)$
Delete	$O(h)$	$O(\lg n)$
Insert	$O(h)$	$O(\lg n)$

h can be very large. But

Search	$O(h)$	$O(n)$
Sort	$O(n)$	$O(n \lg n)$

And there are versions of BST that are self balancing so we can make h always $O(\lg n)$.

Build	$O(n \lg n)$	$O(n)$

7.2

Finally we mention that Red-Black trees are variation on BST that always guarantees that the depth of the tree never exceeds $2\log(n)$.
I encourage you to read Chapter 13 in CLRS.

We now start a new topic called Dynamic Programming.
This is an algorithmic techniques that solves a problem by combining solutions to subproblems.
It is similar to divide and conquer from this perspective but it is different in that it does not divide the problem to disjoint subproblems but rather it may solve overlapping subproblems of small size, use the solutions to solve larger subproblems etc
Dynamic Programming is mainly deployed on optimization problems - these are problems that may have many solutions and we wish to find the optimal one

It is probably best to explain the technique via an example.

Rod cutting.
We have a steel rod of length n. We can sell a piece of the rod of length i (i is integral) for p_i dollars. Our aim is to maximize the revenue.
Notice that no matter how we cut the rod we will always get something. So this is an optimization problem.
Notice further that there are many ways to partition n - roughly $2^{\sqrt{n}}$ ways. Thus it will require too much time to cover all those options.
The way to solve the problem is to compute, recursively, for each $i < n$ the optimal solution r_i, and from it compute r_n.
r_1 is just p_1. Assume now that we have computed $r_1, \ldots, r_{n-1}$ and let us compute r_n.
Observe that $r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{\lfloor n/2 \rfloor} + r_{\lceil n/2 \rceil})$. Another way to express r_n is $r_n = \max\limits_{1 \le i \le n}(p_i + r_{n-i})$ where $r_0 = 0$.

7.3

Indeed, if the length of the first cut in the optimal solution is i then the value of the solution is at most $p_i + r_{n-i}$ and it can clearly be made to have this value.

The nice thing about this is that we always solve the same optimization problem but for smaller input lengths. Such structure is called optimal substructure - optimal solution to a problem relies on optimal solutions to subproblems.

Let us present the pseudocode now. p is an array holding the prices.

If we implement the idea we showed carelessly then we will have

Cut-Rod(p, n)

1. if $n = 0$
2. return 0
3. $q = -\infty$
4. For $i = 1$ to n
5. $q = \max(q, p(i) + \text{Cut-Rod}(p, n-i))$
6. return q

Note that $T(n)$ satisfies $T(n) = \Theta(1) + T(n-1) + \ldots + T(1)$

the solution to this recursion is exponential...

How ever, we don't have to rerun the calculation for ~~T(k)~~ $q(k)$ again and again. Instead we could store that value and use it when necessary.

~~Bu~~

```
Bottom-Up-Cut-Rod(p, n)
let r[0..n] be a new array
r[0] = 0
For j=1 to n
    q = -∞
    For i=1 to j
        q = max(q, p[i] + r[j-i])
    r[j] = q
return r[n]
```

Running time is only $O(n^2)$ now! (due to the nested For loops it is also $\Omega(n^2)$).

The algorithm that we gave computed the best revenue but it doesn't tell us how to actually cut the rod. To construct the solution we just need to add a couple of lines to the code.

```
Extended-Bottom-Up-Rod-Cut(p, n)
Let r[0..n], s[0..n] be new arrays
r[0] = 0
For j=1 to n
    q = -∞
    For i=1 to j
        If q < p[i]+r[j-i]
            q = p[i] + r[j-i]
            s[j] = i
    r[j] = q
Return r, s
```

7.5

The difference is that now $S[j]$ stores the "first" cut we have to do when the input rod has size j.
To output the best partition we simply:

Print-Cut-Rod-Solution(p, n)

1. (r, s) = Extended-Bottom-Up-Cut-Rod(p, n)
2. while n > 0
3. print s[n]
4. n = n - S[n]

Let us see another example.
Longest Common Subsequence:
Given two sequences of symbols from some alphabet $X \in \Sigma^m, Y \in \Sigma^n$ we wish to determine (and output) the longest common subsequence.
E.g. $X = (2, 4, 5, 7)$ $Y = (4376)$ then $(4, 7)$ is a subsequence of both and it is the longest.
This problem has practical significance: it can be used as a measure for comparing files / DNA sequences / Codes etc. i.e. it is a measure of similarity.

How to compute the LCS of X and Y?
Observe that if $x_m = y_n$ then this symbol is in the LCS. Thus we can put it in the LCS and move to computing the LCS of $(X_1 \dots X_{m-1}), (Y_1 \dots Y_{n-1})$.
If $x_m \neq y_n$ then the LCS is either $LCS((X_1 \dots X_{m-1}), Y)$ or $LCS(X, (Y_1 \dots Y_{n-1}))$.
Thus, if we know the LCS of $(X_{1 \dots i}, Y_{1 \dots j})$ for all i, j then we can compute $LCS(X_{1 \dots m}, Y_{1 \dots n})$.

7.6

For simplicity we denote $X_i = (x_1, \ldots, x_i)$ $Y_j = (y_1, \ldots, y_j)$

We define an array $C[i,j]$ that will hold the LCS of X_i, Y_j for $1 \le i \le m$, $1 \le j \le n$

We thus have

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

We can now write down the DP alg.

```
LCS-Length(X, Y)
m = X.length
n = Y.length
For i=1 to m
    C[i,0]=0
For j=0 to n
    C[0,j]=0
For i=1 to m
    For j=1 to n
        If x_i = y_j
            C[i,j] = C[i-1,j-1]+1
        Else If C[i-1,j] ≥ C[i,j-1]
            C[i,j] = C[i-1,j]
        Else C[i,j] = C[i,j-1]
Return C
```

This alg. compute $C[i,j]$ $\forall i,j$. But how do we return the LCS itself?

Note that $C[i,j]$ also tells us which symbol to print!

If $C[m,n] > C[m-1,n], C[m,n-1]$ then we must have included $x_m = y_n$ in the LCS!

Print-LCS(X, c, i, j) (no need for Y now)

1. If $i=0$ or $j=0$
2. return " "
3. If $C[i,j] > C[i,j-1]$ and $C[i,j] > C[i-1,j]$
4. Print-LCS$(X, c, i-1, j-1)$
5. Print X_i
6. Else If $C[i,j] = C[i,j-1]$
7. Print-LCS$(X, c, i, j-1)$
8. Else Print-LCS$(X, c, i-1, j)$

Observe that given c, the running time is $O(i+j)$ as we have a constant number of operations and a call to the same alg. with $(i-1, j-1)$ or $(i-1, j)$ or $(i, j-1)$, i.e. sum of $i+j$ goes down by ≥ 1 at each sub alg.

From these examples we see that a DP alg. follows the following reasoning:

1. Characterize the structure of the optimal solution.
2. Recursively define the value of ~~the~~ an optimal solution
3. Compute the value of an optimal solution, typically in a bottom-up fashion
4. Construct an optimal solution from ~~the~~ computed information.

7.8

The last example we will see is Matrix-Chain-Multiplication.

Assume we wish to compute $A_1 \cdot A_2 \cdot \ldots \cdot A_n$ where A_i has dimension $p_{i-1} \times p_i$
(# columns of A_i = # rows of A_{i+1}). What is the best way to multiply them?
I.e. what is the best i to do $(A_1 \cdot \ldots \cdot A_i) \cdot (A_{i+1} \cdot \ldots \cdot A_n)$?

Let us assume for simplicity the trivial MM alg. that takes ~~$p_{i-1}\, p_i\, p_{i+1}$~~ $k \cdot m \cdot n$ to multiply an $k \times m$ matrix with an $m \times n$ matrix.
(Strassen's alg. is better but for simplicity let us assume the easy alg.)

So how do we solve it?

Step 1: Notice that if the first split is at location k, i.e. we compute $(A_1 \ldots A_k) \cdot (A_{k+1} \ldots A_n)$
then the optimal solution will be composed of optimal solutions to the subproblems.
This is great for DP!

Step 2: Recursive solution. Let $m[i,j]$ denote the cost of computing $A_i \cdot A_{i+1} \cdot \ldots \cdot A_j$.
Then, if $i \le k < j$ is the optimal break point we get
$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} \cdot p_k \cdot p_j$
because $(A_i \cdot \ldots \cdot A_k)$ is $p_{i-1} \times p_k$, $(A_{k+1} \cdot \ldots \cdot A_j)$ is $p_k \times p_j$

$$\Rightarrow m[i,j] = \begin{cases} 0 & i = j \\ \min\limits_{i \le k < j} m[i,k] + m[k+1,j] + p_{i-1} p_k p_j & \text{if } i < j \end{cases}$$

2.9

The alg. is now simple. We will also keep track at each step where the optimal partition is.

```
Matrix-Chain-Order (p)
n = p.length - 1
Let m[1..n, 1..n], s[1...n-1, 2..n] be new tables
For i=1 to n
    m[i,i] = 0
For l = 2 to n                          l is the chain length   l = (i-j+1)
    For i = 1 to n-l+1                  i is start point
        j = i+l-1                       j is end point
        m[i,j] = ∞                      initialization
        For k = i to j-1                k is possible break point
        q = m[i,k] + m[k+1,j] + p_{i-1}·p_k·p_j
        If q < m[i,j]
            m[i,j] = q
            s[i,j] = k
Return m and s.
```

Running time is $O(n^3)$ because of the nested for loops. Here's a nice way to print the Parentheses:

```
Print-optimal-Parens (s, i, j)
if i = j
    Print "A_i"
Else print "("
    Print-Optimal-Parens (s, i, s[i,j])
    Print-Optimal-Parens (s, s[i,j]+1, j)
    Print ")"
```