Fibonacci Sequence (cont)

- Keep a table

```c
int Fibrt(int n, int*Values) {
   // Assume Values has at least n slots, and
   // all slots are initialized to 0
   if (n <= 1) return 1; // Base case
   if (Values[n] == 0) // Compute and store
      Values[n] = Fibrt(n-1, Values) +
                  Fibrt(n-2, Values);
   return Values[n];
}
```

- Cost?
- We don’t need table, only last 2 values. 
  - Key is working bottom up.

Chained Matrix Multiplication

**Problem:** Compute the product of \( n \) matrices

\[ M = M_1 \times M_2 \times \cdots \times M_n \]

as efficiently as possible.

If \( A \) is \( r \times s \) and \( B \) is \( s \times t \), then

\[ \text{COST}(A \times B) = r \times t \]

If \( C \) is \( t \times u \) then

\[ \text{COST}((A \times B) \times C) = t \times u \]

\[ \text{COST}((A \times (B \times C))) = u \times t \]

View as binary trees:

Order Matters

Example:

\[ A = 2 \times 8; B = 8 \times 5; C = 5 \times 20 \]

\[ \text{COST}((A \times B) \times C) = \]

\[ \text{COST}(A \times (B \times C)) = \]

Chained Matrix Induction

**Induction Hypothesis:** We can find the optimal evaluation tree for the multiplication of \( \leq n - 1 \) matrices.

**Induction Step:** Suppose that we start with the tree for:

\[ M_1 \times M_2 \times \cdots \times M_{n-1} \]

and try to add \( M_n \).

Two obvious choices:

- Multiply \( M_{n-1} \times M_n \) and replace \( M_{n-1} \) in the tree with a subtree.
- Multiply \( M_n \) by the result of \( P(n - 1) \): make a new root.

Visually, adding \( M_n \) may radically order the (optimal) tree.

Chained Matrix Induction

**Induction Hypothesis:** We can find the optimal evaluation tree for the multiplication of \( \leq n - 1 \) matrices.

**Induction Step:** Suppose that we start with the tree for:

\[ M_1 \times M_2 \times \cdots \times M_{n-1} \]

and try to add \( M_n \).

Two obvious choices:

- Multiply \( M_{n-1} \times M_n \) and replace \( M_{n-1} \) in the tree with a subtree.
- Multiply \( M_n \) by the result of \( P(n - 1) \): make a new root.

Visually, adding \( M_n \) may radically order the (optimal) tree.
Alternate Induction

**Induction Step:** Pick some multiplication as the root, then recursively process each subtree.
- Which one? Try them all!
- Choose the cheapest one as the answer.
- How many choices?

Observation: If we know the $i$th multiplication is the root, then the left subtree is the optimal tree for the first $i-1$ multiplications and the right subtree is the optimal tree for the last $n-i-1$ multiplications.

Notation: for $1 \leq i \leq n$,
$$c[i,j] = \text{minimum cost to multiply } M_i \times M_{i+1} \times \cdots \times M_j,$$
So,$$c[1,n] = \min_{1 \leq i \leq n-1} n_i + c[i,j] + c[i+1,n].$$

Choose the cheapest one as the answer.

Which one? Try them all!

Dynamic Programming

Make an $n \times n$ table with entry $(i,j) = c[i,j]$.

<table>
<thead>
<tr>
<th></th>
<th>$c[1,1]$</th>
<th>$c[1,2]$</th>
<th>$\cdots$</th>
<th>$c[1,n]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c[2,1]$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\ddots$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
</tr>
<tr>
<td>$c[n,1]$</td>
<td>$\vdots$</td>
<td>$\vdots$</td>
<td>$\ddots$</td>
<td>$c[n,n]$</td>
</tr>
</tbody>
</table>

Only upper triangle is used.

Fill in table diagonal by diagonal.
$c[i,j] = 0$.
For $1 \leq i < j \leq n$,
$$c[i,j] = \min_{i \leq k \leq j-1} n_i r_{ij} + c[i,k] + c[k+1,j].$$

Dynamic Programming Analysis

- The time to calculate $c[i,j]$ is proportional to $j-i$.
- There are $\Theta(n^2)$ entries to fill.
- $T(n) = O(n^3)$.
- Also, $T(n) = \Omega(n^3)$.
- How do we actually find the best evaluation order?

Analysis

**Base Cases:** For $1 \leq k \leq n$, $c[k,k] = 0$.
More generally:
$$c[i,j] = \min_{1 \leq k \leq j-1} n_i r_{ij} + c[i,k] + c[k+1,j].$$

Solving $c[i,j]$ requires $2(j-i)$ recursive calls.

**Analysis:**
$$T(n) = \sum_{k=1}^{n-1} (T(k) + T(n-k)) = 2 \sum_{k=1}^{n-1} T(k)$$
$$T(1) = 1$$
$$T(n+1) = T(n) + 2T(n) = 3T(n)$$
$$T(n) = \Theta(3^n)$$

But there are only $\Theta(n^2)$ values $c[i,j]$ to be calculated!

2 calls for each root choice, with $(j-i)$ choices for root. But, these don’t all have equal cost.

$$T(n+1) = 2 \sum_{i=1}^{n} T(k)$$
So:
$$T(n+1) - T(n) = 2 \sum_{i=1}^{n} T(k) - 2 \sum_{i=1}^{n-1} T(k)$$
$$= 2T(n)$$
$$T(n+1) = 3T(n)$$

Actually, since $j > i$, only about half that needs to be done.

Dynamic Programming

The array is processed starting with the middle diagonal (all zeros), diagonal by diagonal toward the upper left corner.

Dynamic Programming Analysis

For middle diagonal of size $n/2$, each costs $n/2$.

For each $c[i,j]$, remember the $k$ (the root of the tree) that minimizes the expression.
So, store in the table the next place to go.
Summary

- Dynamic programming can often be added to an inductive proof to make the resulting algorithm as efficient as possible.
- Can be useful when divide and conquer fails to be efficient.
- Usually applies to optimization problems.
- Requirements for dynamic programming:
  1. Repeated solution of subproblems
  2. Small number of subproblems, small amount of information to store for each subproblem.
  3. Base case easy to solve.
  4. Easy to solve one subproblem given solutions to smaller subproblems.

Sorting

Each record contains a field called the key.
Linear order: comparison.

The Sorting Problem

Given a sequence of records \( R_1, R_2, \ldots, R_n \) with key values \( k_1, k_2, \ldots, k_n \), respectively, arrange the records into any order such that records \( R_{k_1}, R_{k_2}, \ldots, R_{k_n} \) have keys obeying the property \( k_1 \leq k_2 \leq \ldots \leq k_n \).

Measures of cost:
- Comparisons
- Swaps

Insertion Sort

```c
void inssort(Elem* A, int n) { // Insertion Sort
    for (int i=1; i<n; i++) // Insert i'th record
        for (int j=i; (j>0) && (A[j].key<A[j-1].key); j--)
            swap(A, j, j-1);
}
```

Best Case: \( n-1 \) comparisons.
Worst Case: \( n^2/2 \) swaps and compares.
Average Case: \( n^2/4 \) swaps and compares.

Insertion sort has great best-case performance.

Exchange Sorting

- **Theorem**: Any sort restricted to swapping adjacent records must be \( \Omega(n^2) \) in the worst and average cases.
- **Proof**:
  - For any permutation \( P \), and any pair of positions \( i \) and \( j \), the relative order of \( i \) and \( j \) must be wrong in either \( P \) or the inverse of \( P \).
  - Thus, the total number of swaps required by \( P \) and the inverse of \( P \) MUST be
    \[
    \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}
    \]

\( n^2/4 \) is the average distance from a record to its position in the sorted output.
Quicksort

Divide and Conquer: divide list into values less than pivot and values greater than pivot.

```c
void qsort(Elem* A, int i, int j) { // Quicksort
    int pivotindex = findpivot(A, i, j); // Swap to end
    swap(A, pivotindex, j); // Put pivot in place
    int k = partition(A, i-1, j, A[j].key);
    swap(A, k, j); // Sort right
    if ((k-i) > 1) qsort(A, i, k-1); // Sort left
    if ((j-k) > 1) qsort(A, k+1, j); // Sort right
}
```

```c
int findpivot(Elem* A, int i, int j){ return (i+j)/2; }
```

Quicksort Partition

```c
int partition(Elem* A, int l, int r, int pivot) { do { // Move bounds inward until they meet
    while (A[++l].key < pivot); // Move right
    while (r && (A[--r].key > pivot)); // Left
    swap(A, l, r); // Swap out-of-place vals
} while (l < r); // Stop when they cross
return l; // Return first position in right
```

The cost for Partition is $\Theta(n)$.

Partition Example

Initial

<table>
<thead>
<tr>
<th>Initial</th>
<th>l</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>72 6 57 88 85 42 83 73 48 60</td>
<td>l</td>
<td>r</td>
</tr>
<tr>
<td>Pass 1</td>
<td>72 6 57 88 85 42 83 73 48 60</td>
<td>l</td>
</tr>
<tr>
<td>Swap 1</td>
<td>48 6 57 88 85 42 83 73 72 60</td>
<td>l</td>
</tr>
<tr>
<td>Pass 2</td>
<td>48 6 57 88 85 42 83 73 72 60</td>
<td>l</td>
</tr>
<tr>
<td>Swap 2</td>
<td>48 6 57 42 88 83 73 72 60</td>
<td>l</td>
</tr>
<tr>
<td>Pass 3</td>
<td>48 6 57 42 88 83 73 72 60</td>
<td>l</td>
</tr>
</tbody>
</table>

Quicksort Example

```
72 6 57 88 60 42 83 73 48 85
Pivot = 60
```
```
48 6 57 42 88 83 72 73 85 85
Pivot = 42
```
```
42 48 57 60 85 83 88
Pivot = 42
```
```
2014-02-06
```

The cost for Quicksort is $\Theta(n\log n)$. The average case time complexity is $\Theta(n\log n)$. The worst case time complexity is $\Theta(n^2)$. When selecting the pivot, we can use a random pivot to reduce the worst case time complexity.
Cost for Quicksort

Best Case: Always partition in half.

Worst Case: Bad partition.

Average Case:
\[ f(n) = -n + 1 + \frac{1}{n} \sum_{i=0}^{n-1} (f(i) + f(n - i - 1)) \]

Optimizations for Quicksort:
- Better pivot.
- Use better algorithm for small sublists.
- Eliminate recursion.
- Best: Don’t sort small lists and just use insertion sort at the end.

Since the two halves of the summation are identical,

\[ f(n) = 0 \text{ for } n \leq 1 \]
\[ f(n) = n - 1 + \frac{1}{n} \sum_{i=0}^{n-1} f(i) \text{ for } n > 1 \]

QuickSort Average Cost

\[ f(n) = \begin{cases} 
0 & \text{if } n \leq 1 \\
1 & \text{if } n > 1 
\end{cases} \]

Since the two halves of the summation are identical,

\[ f(n) = 0 \text{ for } n \leq 1 \]
\[ f(n) = n - 1 + \frac{1}{n} \sum_{i=0}^{n-1} f(i) \text{ for } n > 1 \]

Multiplying both sides by \( n \) yields

\[ nf(n) = n(n - 1) + 2 \sum_{i=0}^{n-1} f(i) \]

Average Cost (cont.)

Get rid of the full history by subtracting \( nf(n) \) from \( (n + 1)f(n + 1) \)

\[ nf(n) = n(n - 1) + 2 \sum_{i=1}^{n-1} f(i) \]
\[ (n + 1)f(n + 1) = (n + 1)n + 2 \sum_{i=1}^{n} f(i) \]
\[ (n + 1)f(n + 1) - nf(n) = 2n + 2f(n) \]
\[ (n + 1)f(n + 1) = 2n + (n + 2)f(n) \]
\[ f(n + 1) = \frac{2n}{n + 1} + \frac{n + 2}{n + 1} f(n). \]

Average Cost (cont.)

Note that \( \frac{2n}{n + 1} \leq 2 \) for \( n \geq 1 \).

Expand the recurrence to get:

\[ f(n + 1) \leq 2 \cdot \frac{n + 2}{n + 1} f(n) \]
\[ = 2 \cdot \frac{n + 2}{n + 1} \left( 2 + \frac{n + 1}{n} f(n - 1) \right) \]
\[ = 2 \cdot \frac{n + 2}{n + 1} \left( 2 + \frac{n + 1}{n} \left( 2 + \frac{n - 1}{n - 1} f(n - 2) \right) \right) \]
\[ = 2 \cdot \frac{n + 2}{n + 1} \left( 2 + \cdots + \frac{4}{3} \left( 2 + \frac{3}{2} f(1) \right) \right) \]

This optimization means, for list threshold \( T \), that no element is more than \( T \) positions from its destination. Thus, insertion sort’s best case is nearly realized. Cost is at worst \( nT \).
Average Cost (cont.)

\[ f(n+1) \leq 2 \left( \frac{n+2}{n+1} + \frac{n}{n+1} + \cdots \right) \]
\[ + \frac{n+2}{n} + \frac{n+1}{n} + \cdots + \frac{3}{2} \]
\[ = 2 \left( 1 + (n+2) \left( \frac{1}{n+1} + \frac{1}{n} + \cdots + \frac{1}{2} \right) \right) \]
\[ = 2 + 2(n+2)(H_{n+1} - 1) \]
\[ = \Theta(n \log n). \]

\[ H_{n+1} = \Theta(\log n) \]