Fibonacci Sequence (cont)

- Keep a table

```
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.

Fibonacci Sequence (cont)
Keep a table
Cost?
We don't need table, only last 2 values.
Key is working bottom up.

no notes

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
COST$(A \times B) =$
SIZE$(A \times B) =$

If C is $t \times u$ then
COST$((A \times B) \times C) =$
COST$((A \times (B \times C))) =$

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
COST$(A \times B) =$
SIZE$(A \times B) =$
If C is $t \times u$ then
COST$((A \times B) \times C) =$
COST$((A \times (B \times C))) =$

$A \times B$:
COST: rst
SIZE: $r \times t$

$rst + (r \times t)(t \times u) = rst + rtu$.
$(r \times s)[(s \times t)(t \times u)] = (r \times s)(s \times u)$.
$rsu + stu$.

Order Matters

Example:

$$A = 2 \times 8; B = 8 \times 5; C = 5 \times 20$$

COST$((A \times B) \times C) =$
COST$(A \times (B \times C)) =$

View as binary trees:

Order Matters
Example:
$A = 2 \times 8; B = 8 \times 5; C = 5 \times 20$
COST$((A \times B) \times C) =$
COST$(A \times (B \times C)) =$
View as binary trees:

$2 \cdot 8 \cdot 5 + 2 \cdot 5 \cdot 20 = 280$.
$8 \cdot 5 \cdot 20 + 2 \cdot 8 \cdot 20 = 1120$.

Tree for $((A \times B) \times C) =: \cdot \cdot ABC$
Tree for $(A \times (B \times C) =: \cdot A \cdot BC$

We would like to find the optimal order for computation before actually doing the matrix multiplications.

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:
1. Multiply $M_{n-1} \times M_n$ and replace M_{n-1} in the tree with a subtree.
2. 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:
1. Multiply $M_{n-1} \times M_n$ and replace M_{n-1} in the tree with a subtree.
2. Multiply M_n by the result of $P(n-1)$: make a new root.
Visually, adding M_n may radically order the (optimal) tree.

Problem: There is no reason to believe that either of these yields the optimal ordering.

Alternate Induction

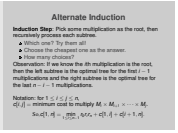
Induction Step: Pick some multiplication as the root, then recursively process each subtree.

- Which one? Try them all!
- Choose the cheapest one as the answer.
- How many choices?

Observation: If we know the ith 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 $1 \le i \le j \le n$,
$c[i,j]$ = minimum cost to multiply $M_i \times M_{i+1} \times \cdots \times M_j$.

$$\text{So, } c[1,n] = \min_{1 \le i \le n-1} r_0 r_i r_n + c[1,i] + c[i+1,n].$$



$n-1$ choices for root.

Analysis

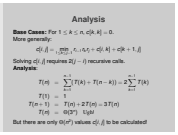
Base Cases: For $1 \le k \le n$, $c[k,k] = 0$.
More generally:

$$c[i,j] = \min_{1 \le k \le j-1} r_{i-1} r_k r_j + c[i,k] + c[k+1,j]$$

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

$$\begin{aligned} T(n) &= \sum_{k=1}^{n-1} (T(k) + T(n-k)) = 2\sum_{k=1}^{n-1} T(k) \\ T(1) &= 1 \\ T(n+1) &= T(n) + 2T(n) = 3T(n) \\ T(n) &= \Theta(3^n) \quad \text{Ugh!} \end{aligned}$$

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:

$$\begin{aligned} T(n+1) - T(n) &= 2\sum_{i=1}^{n} T(k) - 2\sum_{i=1}^{n-1} T(k) \\ &= 2T(n) \\ T(n+1) &= 3T(n) \end{aligned}$$

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

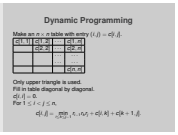
Dynamic Programming

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

$c[1,1]$	$c[1,2]$	$\cdots$	$c[1,n]$
	$c[2,2]$	$\cdots$	$c[2,n]$
		$\cdots$	$\cdots$
		$\cdots$	$\cdots$
			$c[n,n]$

Only upper triangle is used.
Fill in table diagonal by diagonal.
$c[i,i] = 0$.
For $1 \le i < j \le n$,

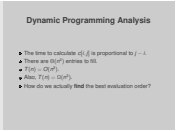
$$c[i,j] = \min_{i \le k \le j-1} r_{i-1} r_k r_j + c[i,k] + c[k+1,j].$$



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

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?



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.

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.

no notes

Sorting

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

The Sorting Problem

Given a sequence of records $R_1, R_2, ..., R_n$ with key values $k_1, k_2, ..., k_n$, respectively, arrange the records into any order s such that records $R_{s_1}, R_{s_2}, ..., R_{s_n}$ have keys obeying the property $k_{s_1} \leq k_{s_2} \leq ... \leq k_{s_n}$.

Measures of cost:

- Comparisons
- Swaps

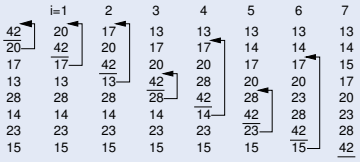
Sorting
Each record contains a field called the **key**.
Linear order: comparison.
The Sorting Problem
Given a sequence of records $R_1, R_2, ..., R_n$ with key values $k_1, k_2, ..., k_n$, respectively, arrange the records into any order s such that records $R_{s_1}, R_{s_2}, ..., R_{s_n}$ have keys obeying the property $k_{s_1} \leq k_{s_2} \leq ... \leq k_{s_n}$.
Measures of cost:
- Comparisons
- Swaps

Linear order means: $a < b$ and $b < c \Rightarrow a < c$.

More simply, sorting means to put keys in ascending order.

Insertion Sort

```
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:
Worst Case:
Average Case:

Insertion Sort
Best Case:
Worst Case:
Average Case:

Best case is 0 swaps, $n - 1$ comparisons.
Worst case is $n^2/2$ swaps and compares.
Average case is $n^2/4$ 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

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}.$$

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
$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

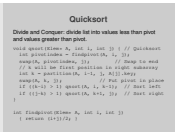
$n^2/4$ is the average distance from a record to its position in the sorted output.

Quicksort

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

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

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

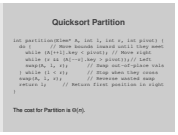



Initial call: `qsort(array, 0, n-1);`

Quicksort Partition

```
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
  swap(A, l, r);         // Reverse wasted swap
  return l;     // Return first position in right
}
```

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



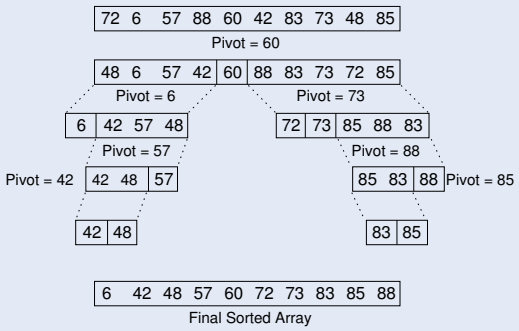
no notes

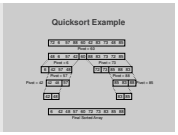
Partition Example

Initial	72	6	57	88	85	42	83	73	48	60	
	l									r	
Pass 1	72	6	57	88	85	42	83	73	48	60	
		l							r		
Swap 1	48	6	57	88	85	42	83	73	72	60	
		l							r		
Pass 2	48	6	57	88	85	42	83	73	72	60	
				l		r					
Swap 2	48	6	57	42	85	88	83	73	72	60	
					l	r					
Pass 3	48	6	57	42	85	88	83	73	72	60	
					l,r						

no notes

Quicksort Example





no notes

Cost for Quicksort

Best Case: Always partition in half.

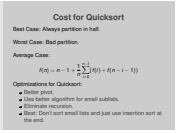
Worst Case: Bad partition.

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.



Think about when the partition is bad. Note the FindPivot function that we used is pretty good, especially compared to taking the first (or last) value.
Also, think about the distribution of costs: Line up all the permuations from most expensive to cheapest. How many can be expensive? The area under this curve must be low, since the average cost is $\Theta(n \log n)$, but some of the values cost $\Theta(n^2)$. So there can be VERY few of the expensive ones.

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.

Quicksort Average Cost

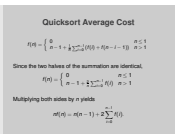
$$f(n) = \begin{cases} 0 & n \leq 1 \\ n - 1 + \frac{1}{n}\sum_{i=0}^{n-1}(f(i) + f(n-i-1)) & n > 1 \end{cases}$$

Since the two halves of the summation are identical,

$$f(n) = \begin{cases} 0 & n \leq 1 \\ n - 1 + \frac{2}{n}\sum_{i=0}^{n-1} f(i) & n > 1 \end{cases}$$

Multiplying both sides by n yields

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



This is a "recurrence with full history".

Think about what the pieces correspond to.
To do Quicksort on an array of size n, we must:

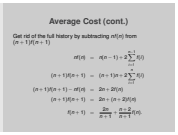
- Partation: Cost n
- Findpivot: Cost c
- Do the recursion: Cost dependent on the pivot's final position.

These parts are modeled by the equation, including the average over all the cases for position of the pivot.

Average Cost (cont.)

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

$$\begin{aligned} nf(n) &= n(n-1) + 2\sum_{i=1}^{n-1} f(i) \\ (n+1)f(n+1) &= (n+1)n + 2\sum_{i=1}^{n} f(i) \\ (n+1)f(n+1) - nf(n) &= 2n + 2f(n) \\ (n+1)f(n+1) &= 2n + (n+2)f(n) \\ f(n+1) &= \frac{2n}{n+1} + \frac{n+2}{n+1}f(n). \end{aligned}$$

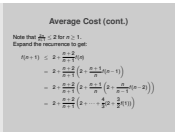


no notes

Average Cost (cont.)

Note that $\frac{2n}{n+1} \leq 2$ for $n \geq 1$.
Expand the recurrence to get:

$$\begin{aligned} f(n+1) &\leq 2 + \frac{n+2}{n+1}f(n) \\ &= 2 + \frac{n+2}{n+1}\left(2 + \frac{n+1}{n}f(n-1)\right) \\ &= 2 + \frac{n+2}{n+1}\left(2 + \frac{n+1}{n}\left(2 + \frac{n}{n-1}f(n-2)\right)\right) \\ &= 2 + \frac{n+2}{n+1}\left(2 + \cdots + \frac{4}{3}(2 + \frac{3}{2}f(1))\right) \end{aligned}$$



no notes

Average Cost (cont.)

$$\begin{aligned} f(n+1) &\leq 2\left(1+\frac{n+2}{n+1}+\frac{n+2}{n+1}\frac{n+1}{n}+\cdots \right. \\ &\qquad \left. +\frac{n+2}{n+1}\frac{n+1}{n}\cdots\frac{3}{2}\right) \\ &= 2\left(1+(n+2)\left(\frac{1}{n+1}+\frac{1}{n}+\cdots+\frac{1}{2}\right)\right) \\ &= 2+2(n+2)\left(\mathcal{H}_{n+1}-1\right) \\ &= \Theta(n\log n). \end{aligned}$$

Average Cost (cont.)

$\mathcal{H}_{n+1} = \Theta(\log n)$