04-630
Data Structures and Algorithms for Engineers

Lecture 19: Algorithm Design Strategies I

Adopted and Adapted from Material by:

David Vernon: vernon@cmu.edu ; www.vernon.eu

Agenda

- Classes of algorithms
  - Iteration
  - Recursion
  - Brute force
  - Divide and conquer
  - Greedy algorithms
  - Dynamic programming
  - Combinatorial search and backtracking
  - Branch and bound

Rules of Thumb: algorithm design

1. Handle repetitive tasks through iteration. [resource limited situations]

Rules of Thumb: algorithm design

1. Handle repetitive tasks through iteration. [resource limited situations]

2. Iterate elegantly though recursion. [elegance & simplicity]

Rules of Thumb: algorithm design

1. Handle repetitive tasks through iteration. [resource limited situations]

2. Iterate elegantly though recursion. [elegance & simplicity]

3. Use brute force when you are lazy but powerful. [computationally expensive even for small input sizes]

Rules of Thumb: algorithm design

1. Handle repetitive tasks through iteration. [resource limited situations]

2. Iterate elegantly though recursion. [elegance & simplicity]

3. Use brute force when you are lazy but powerful. [computationally expensive even for small input sizes]

4. Test bad options then backtrack. [adds intelligence to brute force]

Rules of Thumb: algorithm design

1. Handle repetitive tasks through iteration. [resource limited situations]

2. Iterate elegantly though recursion. [elegance & simplicity]

3. Use brute force when you are lazy but powerful. [computationally expensive even for small input sizes]

4. Test bad options then backtrack. [adds intelligence to brute force]

5. Save time with heuristics for a reasonable way out.

Rules of Thumb: algorithm design

1. Handle repetitive tasks through iteration. [resource limited situations]

2. Iterate elegantly though recursion. [elegance & simplicity]

3. Use brute force when you are lazy but powerful. [computationally expensive even for small input sizes]

4. Test bad options then backtrack. [adds intelligence to brute force]

5. Save time with heuristics for a reasonable way out.

6. Divide and conquer your toughest opponents.

Rules of Thumb: algorithm design

1. Handle repetitive tasks through iteration. [resource limited situations]
2. Iterate elegantly though recursion. [elegance & simplicity]
3. Use brute force when you are lazy but powerful. [computationally expensive even for small input sizes]
4. Test bad options then backtrack. [adds intelligence to brute force]
5. Save time with heuristics for a reasonable way out.
6. Divide and conquer your toughest opponents.
7. Identify old issues dynamically not to waste energy again.

Rules of Thumb: algorithm design

1. Handle repetitive tasks through iteration. [resource limited situations]
2. Iterate elegantly though recursion. [elegance & simplicity]
3. Use brute force when you are lazy but powerful. [computationally expensive even for small input sizes]
4. Test bad options then backtrack. [adds intelligence to brute force]
5. Save time with heuristics for a reasonable way out.
6. Divide and conquer your toughest opponents.
7. Identify old issues dynamically not to waste energy again.
8. Bound your problem so the solution doesn't escape.

F. F. Wladston, Computer Science Distilled: Learn the Art of Solving Computational Problems. Code Energy LLC (2017)

Rules of Thumb: algorithm design

1. Handle repetitive tasks through iteration. [resource limited situations]
2. Iterate elegantly though recursion. [elegance & simplicity]
3. Use brute force when you are lazy but powerful. [computationally expensive even for small input sizes]
4. Test bad options then backtrack. [adds intelligence to brute force]
5. Save time with heuristics for a reasonable way out.
6. Divide and conquer your toughest opponents.
7. Identify old issues dynamically not to waste energy again.
8. Bound your problem so the solution doesn't escape.

F. F. Wladston, Computer Science Distilled: Learn the Art of Solving Computational Problems. Code Energy LLC (2017)

Iteration & Recursion

- ***Iteration***: Uses loops to repeat a process until a condition is met.

```
factorialIteration(int n):
        int result=1;
        for i=1 to n:
                result=result*i
        return result
```

- ***Recursion***: achieves repetition through function calls.
  - Uses a ***base case*** to ensure the function returns.

```
factorialRecursion(int n):
        if n=0 or n=1 then
                return 1 #base case
        return n*factorialRecursion(n-1)
```

Agenda

- Classes of algorithms
  - Brute force
  - Divide and conquer
  - Greedy algorithms
  - Dynamic programming
  - Combinatorial search and backtracking
  - Branch and bound

Brute Force [Complete or Exhaustive Search]

- Brute force is a straightforward approach to solve a problem based on a simple formulation of problem
- Often ***without any deep analysis*** of the problem
- Perhaps the easiest approach to apply and is useful for solving ***small-size instances of a problem***
- May result in ***naïve solutions*** with ***poor performance***

Brute Force

Some examples of brute force algorithms are:

- Computing a^n ($a > 0$, n a non-negative integer) by repetitive multiplication: a x a x … x a
  - For a more efficient approach, see https://en.wikipedia.org/wiki/Exponentiation_by_squaring
- Computing n! by repetitive multiplication: n x n-1 x n-2, …
  - For more efficient approaches, see http://www.luschny.de/math/factorial/FastFactorialFunctions.htm
- Sequential (linear) search
- Selection sort, Bubble sort

Brute Force

Maximum sub-array problem / Grenander's Problem

– Given a sequence of integers $i_1, i_2, \ldots, i_n$, find the ***sub-sequence (a contiguous sub-array)*** with the maximum sum

  • If all numbers are negative the result is 0 (Why?)

– Examples:

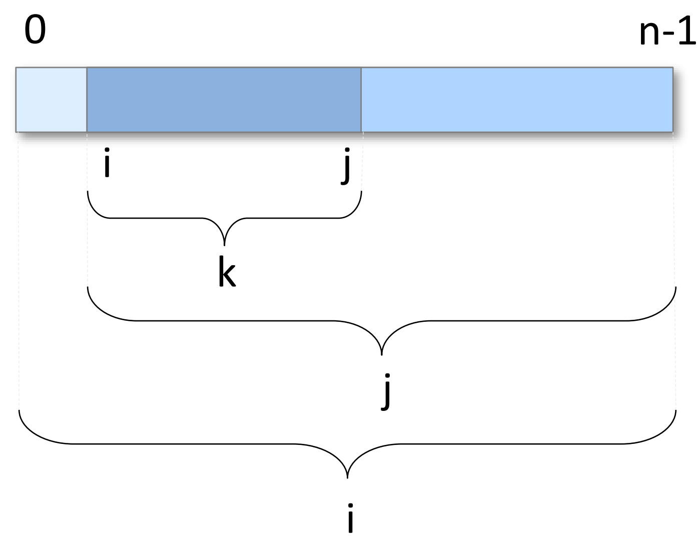
  -2, **11, -4, 13**, -4, 2 has the solution 20

  1, -3, **4, -2, -1, 6** has the solution 7

Brute Force

Maximum subarray problem: brute force solution $O(n^3)$

```
int grenanderBF(int a[], int n) {
   int maxSum = 0;
   for (int i = 0; i < n; i++) {
      for (int j = i; j < n; j++) {
         int thisSum = 0;
         for (int k = i; k <= j; k++) {
            thisSum += a[ k ];
         }
         if (thisSum > maxSum) {
            maxSum = thisSum;
         }
      }
   }
   return maxSum;
}
```



Brute Force

Maximum sub-array problem

- Divide and Conquer algorithm $O(n \log n)$
- Kadane's algorithms $O(n)$ … dynamic programming

Agenda

- Classes of algorithms
  - Brute force
  - Divide and conquer
  - Greedy algorithms
  - Dynamic programming
  - Combinatorial search and backtracking
  - Branch and bound

Divide and Conquer

- Divide-and conquer (D&Q)
  - Given an instance of the problem
  - Divide this into smaller sub-instances (often two)
  - Independently solve each of the sub-instances
  - Combine the sub-instance solutions to yield a solution for the original instance
- With the D&Q method, the size of the problem instance is reduced by a factor (e.g. half the input size)

Divide and Conquer

```
// Generic Divide and Conquer Algorithm

divideAndConquer(Problem p) {
   if (p is simple or small enough) {
      return simpleAlgorithm(p);
   } else {
      divide p in smaller instances p1, p2, ..., pn
      Solution solutions[n];
      for (int i = 0; i < n; i++) {
         solutions[i] = divideAndConquer(pi);
      }
      return combine(solutions);
   }
}
```

Divide and Conquer

- Often yield a recursive formulation

- Examples of D&Q algorithms
  - Quicksort algorithm
  - Mergesort algorithm
  - Fast Fourier Transform

Divide and Conquer

Mergesort

UNSORTEDSEQUENCE

UNSORTED SEQUENCE

UNSO RTED SEQU ENCE

UN SO RT ED SE QU EN CE

NU OS RT DE ES QU EN CE

NOSU DERT EQSU CEEN

DENORSTU CEEENQSU

CDEEEENNOQRSSTUU

Divide and Conquer

```
void mergesort(Item a[], int l, int r) {
    if (l>=r) {
        return;
    } else {
        int m = (r + l) / 2;
        mergesort(a, l, m);
        mergesort(a, m+1, r);
        merge(a, l, m, r);
    }
}
```

Already sorted?

Divide the list into two equal parts

Sort the two halves recursively

Merge the sorted halves into a sorted whole

```
void mergesort(Item a[], int size) {
    mergesort(a, 0, size-1);
}
```

Divide and Conquer

```
int grenanderDQ(int a[], int l, int h) {
    if (l > h) return 0;
    if (l = h) return max(0, a[l]);
    int m = (l + h) / 2;
    int sum = 0;
    int maxLeft = 0;
    for (int i = m; i >= l; i--) {
        sum += a[i];
        maxLeft = max(maxLeft, sum);
    }
    sum = 0;
    int maxRight = 0;
    for (int i = m + 1; i <= h; i++) {
        sum += a[i];
        maxRight = max(maxRight, sum);
    }
    int maxL = grenanderDQ(a, l, m);
    int maxR = grenanderDQ(a, m+1, h);
    int maxC = maxLeft + maxRight;
    return max(maxC, max(maxL, maxR));
}
```

Divide the problem

Solve the sub-problem

Solve the sub-problems

Solve the sub-problem

Combine the solutions

Agenda

- Classes of algorithms
  - Brute force
  - Divide and conquer
  - Greedy algorithms
  - Dynamic programming
  - Combinatorial search and backtracking
  - Branch and bound

Greedy Algorithms

- Try to find solutions to problems step-by-step
  - A partial solution is incrementally expanded towards a complete solution
  - In each step, there are several ways to expand the partial solution
  - The best alternative for the moment is chosen, the others are discarded
- At each step the choice must be **locally optimal** – this is the central point of this technique

Greedy Algorithms

- Examples of problems that can be solved using a greedy algorithm:
  - Finding the minimum spanning tree of a graph (Prim’s algorithm)
  - Finding the shortest distance in a graph (Dijkstra’s algorithm)
  - Using Huffman trees for optimal encoding of information
  - The Knapsack problem

Agenda

- Classes of algorithms
  - Brute force
  - Divide and conquer
  - Greedy algorithms
  - Dynamic programming
  - Combinatorial search and backtracking
  - Branch and bound

Dynamic Programming

- Dynamic programming is similar to D&Q
  - Divides the original problem into smaller sub-problems
- Sometimes it is hard to know beforehand which sub-problems are needed to be solved in order to solve the original problem
- Dynamic programming solves a large number of sub-problems
- … and uses *some* of the sub-solutions to form a solution to the original problem

Dynamic Programming

- In an optimal sequence of choices, actions or decisions for each sub-sequence must also be optimal:
  - An optimal solution to a problem is a combination of optimal solutions to some of its sub-problems
  - Not all optimization problems adhere to this principle

Dynamic Programming

- One disadvantage of using D&Q is that the process of recursively solving separate sub-instances can result in **the same computations being performed repeatedly**
  - A condition called overlapping subproblems.

- The idea behind dynamic programming is to avoid calculating the same quantity twice, usually by **maintaining a table of sub-instance results**

Dynamic Programming

- The same sub-problems may reappear
- To avoid solving the same sub-problem more than once, sub-results are saved in a data structure that is updated dynamically
- Sometimes the result structure (or parts of it) may be computed beforehand
- Strategy:
  - reuse partial calculations (a process called memoization).
  - ***memoization***: an optimization technique that stores the results of expensive function calls and returns the cached result when the same inputs occur again.
    - Speeds up an algorithm.

Dynamic Programming

There are three steps involved in solving a problem by dynamic programming:

1. Formulate the answer as a **recurrence relation** or **recursive algorithm**

2. Show that the number of different parameter values taken on by your recurrence is bounded by a (hopefully small) polynomial---storage reasons.

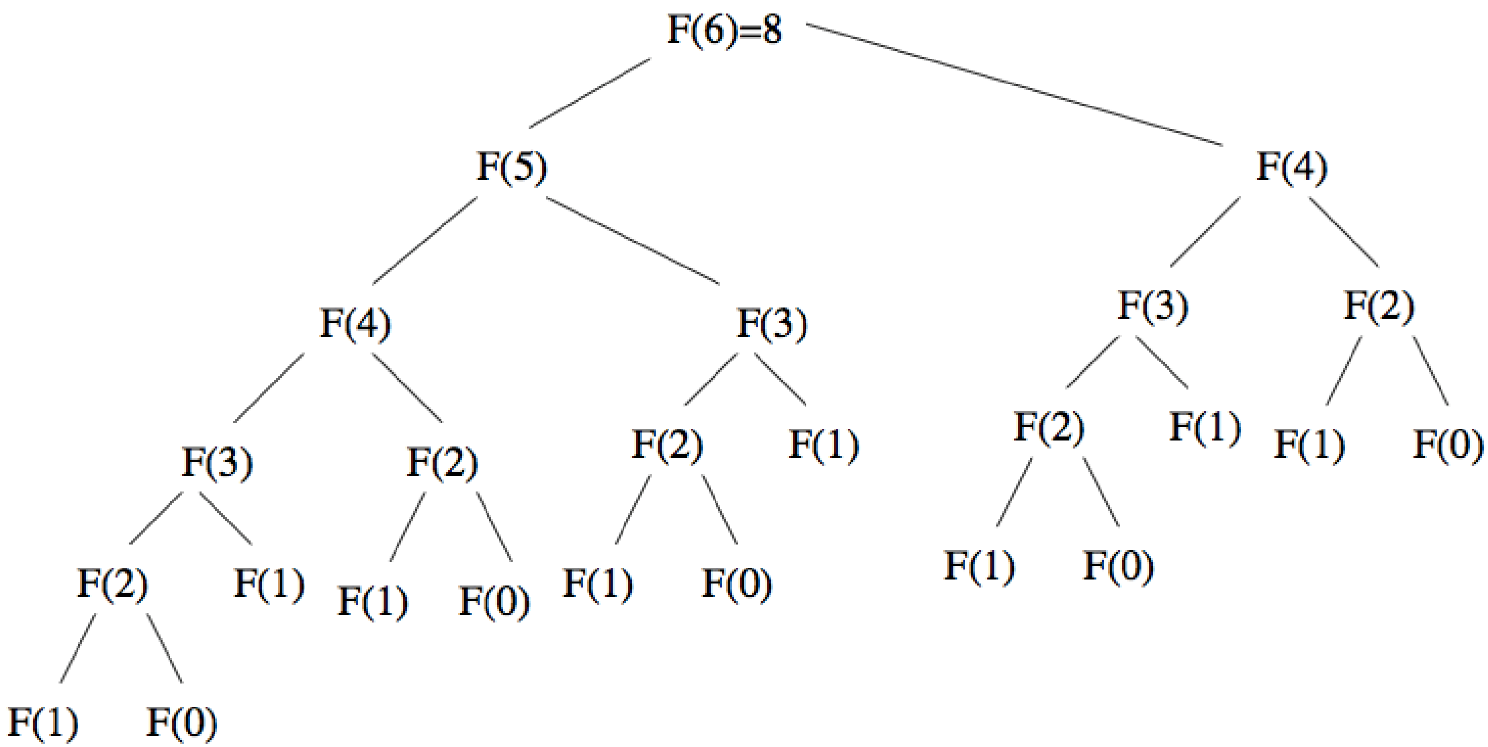
3. Specify an order of evaluation for the recurrence so the partial results you need are always available when you need them

Dynamic Programming

```
/* fibonacci by recursion O(1.618^n) time complexity */

long fib_r(int n) {
   if (n == 0)
      return(0);
   else
      if (n == 1)
         return(1);
      else
         return(fib_r(n-1) + fib_r(n-2));
}
```

```
fib_r(4)  → fib(3) + fib(2)
          → fib(2) + fib(1) + fib(2)
          → fib(1) + fib(0) + fib(1) + fib(2)
          → fib(1) + fib(0) + fib(1) + fib(1) + fib(0)
```

Dynamic Programming: setting(Fibonacci by recursion)



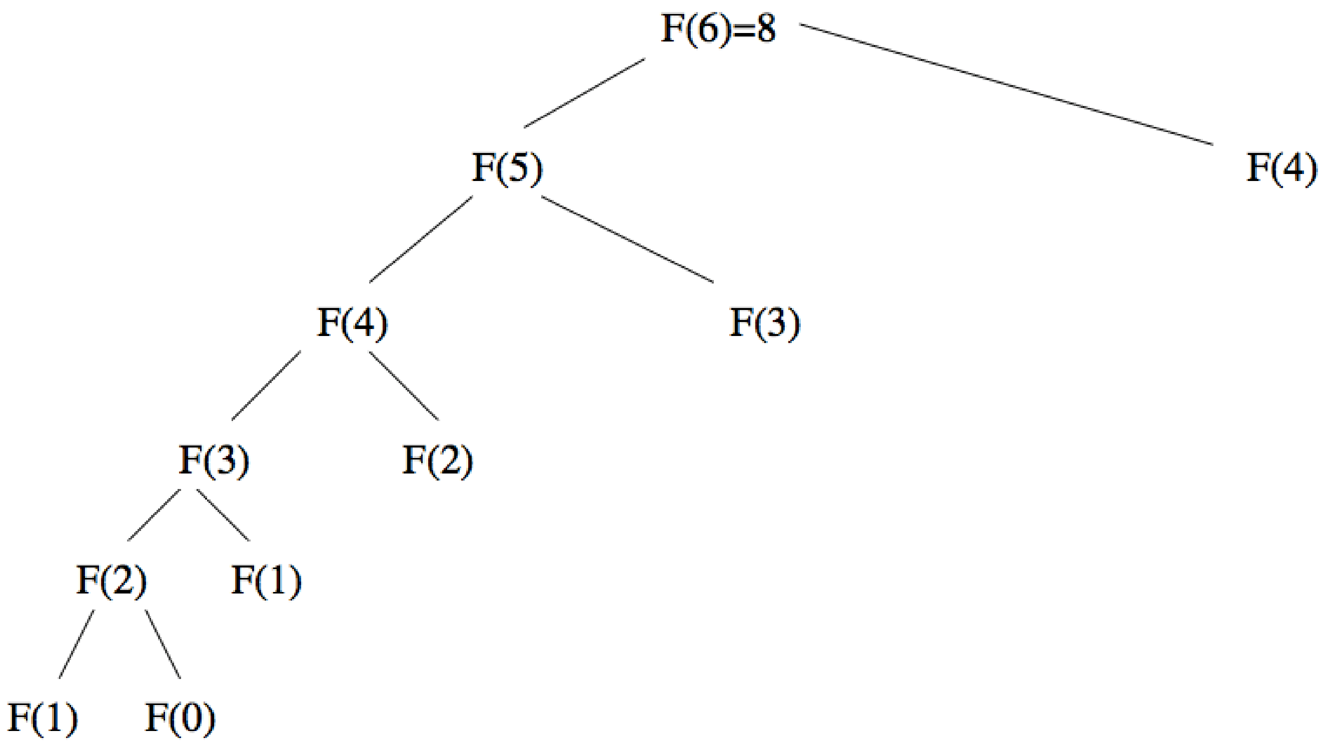
Dynamic Programming: Fibonacci with memoization

- #data structure for memoization
- set m_fib to first two Fibonacci numbers #0 and 1
- function fibonacci(n):
  - if n not in m_fib then #this is how the computation speed is boosted
    - #performs recursion and memoization---caching
    - m_fib[n]=fibonacci(n-1)+fibonacci(n-2)
  - return m_fib[n]

Dynamic Programming

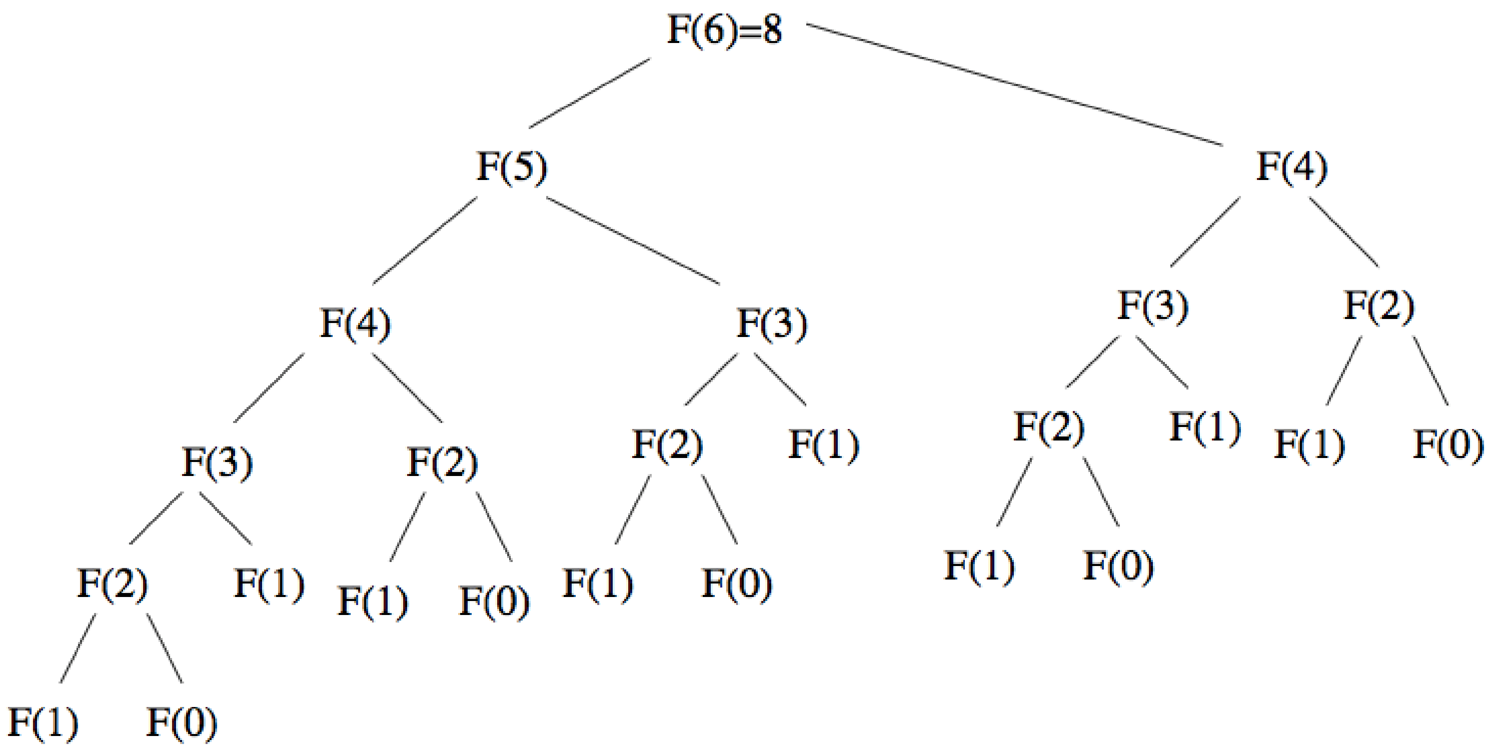
```
#define MAXN 45      /* largest interesting n                  */
#define UNKNOWN -1   /* contents denote an empty cell          */
long f[MAXN+1];      /* array for caching computed fib values  */

/* fibonacci by caching: O(n) storage & O(n) time              */

long fib_c(int n) {
   if (f[n] == UNKNOWN)
      f[n] = fib_c(n-1) + fib_c(n-2);
   return(f[n]);
}

long fib_c_driver(int n) {
   int i; /* counter */

   f[0] = 0;
   f[1] = 1;
   for (i=2; i<=n; i++)  //Careful with array indexing---<n or <=n?
      f[i] = UNKNOWN;
   return(fib_c(n)); //n or n-1?
}
```

Dynamic Programming: recursive tree



Notice that we only need to perform ***one computation*** for F(4), F(3), and F(2). This is evident from what appears like a DFS expansion.

Dynamic Programming: Compare with classical Recursion



Notice that we need to perform:

- ***two computations*** for F(4),
- three for F(3), and
- five for F(2).

Dynamic programming: Fibonacci- caching & iteration

- #data structure for memoization
- function fibonacci(n):
  - set m_fib to first two Fibonacci numbers #0 and 1
  - for i=2 to n
    - #performs memoization---caching
    - m_fib[n]=m_fib[n-1]+m_fib[n-2]
  - return m_fib[n]

Dynamic Programming

```
/* fibonacci by dynamic programming: cache & no recursion          */
/* NB: need correct order of evaluation in the recurrence relation  */
/* O(n) storage & O(n) time                                         */


long fib_dp(int n) {
   int i; /* counter */
   long f[MAXN+1]; /* array to cache computed fib values */

   f[0] = 0;
   f[1] = 1;

   for (i=2; i<=n; i++)
      f[i] = f[i-1]+f[i-2];

   return(f[n]);//array indexing considerations….
}
```

Dynamic Programming

```
/* fibonacci by dynamic programming: minimal cache & no recursion        */
/* O(1) storage & O(n) time                                              */

long fib_ultimate(int n) {

   int i;                     /* counter */
   long back2=0, back1=1;     /* last two values of f[n] */
   long next;                 /* placeholder for sum */

   if (n == 0) return (0);

   for (i=2; i<n; i++) {
      next = back1+back2;
      back2 = back1;
      back1 = next;
   }
   return(back1+back2); //covers n==1 as well
}
```

Summary

- *Several strategies* exist.

- *Choice* of strategy should be guided by:
  - *available resources* (CPU, memory etc.),
  - *problem size*, and
  - *time complexity*.

- Ultimate *goal* is to develop *optimal solutions*.

Next

- Classes of algorithms
  - Brute force
  - Divide and conquer
  - Greedy algorithms
  - Dynamic programming
  - Combinatorial search and backtracking
  - Branch and bound