Algorithms and Data Structures
CS-CO-412

David Vernon
Professor of Informatics
University of Skövde
Sweden

david@vernon.eu
www.vernon.eu
Algorithmic Strategies

Lecture 15
Topic Overview

- Brute-force
- Divide-and-conquer
- Greedy algorithms
- Dynamic Programming
- Combinatorial Search & Backtracking
- Branch-and-bound
Brute Force

- Brute force is a straightforward approach to solve a problem based on a simple formulation of the 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.
- Some examples of brute force algorithms are:
  - Computing $a^n$ ($a > 0$, $n$ a non-negative integer) by repetitive multiplication: $a \times a \times \ldots \times a$
  - Computing $n!$
  - Sequential search
  - Selection sort, Bubble sort
Brute Force

• Maximum subarray problem / Grenander’s Problem
  
  – Given a sequence of integers $i_1, i_2, \ldots, i_n$, find the sub-sequence with the maximum sum
  
  • If all numbers are negative the result is 0
  
  – Examples:

    -2, 11, -4, 13, -4, 2 gives the solution 20

    1, -3, 4, -2, -1, 6 gives the solution 7
Brute Force

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

```c
int gernanderBF(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 subarray problem
  – Divide and Conquer algorithm $O(n \log n)$
  – Kadane’s algorithms $O(n)$ … dynamic programming
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

• Often yield a recursive formulation

• Examples of D&Q algorithms
  – Quicksort algorithm
  – Mergesort algorithm
  – Fast Fourier Transform
Divide and Conquer

Mergesort

UNSORTEDSEQUENCE

UNSORTED

UNSO

RTED

SEQUENCE

SEQUENCE

ENCE

UNOSRTEDSEQUENCE

CDEEEENNOQRSSTUU
void mergesort(Item a[], int l, int r) {
  if (r-l <= 1) {
    return;                
    // Already sorted?
  } else {
    int m = (r + l) / 2;
    mergesort(a, l, m);
    mergesort(a, m+1, r);
    merge(a, l, m, r);      
    // Merge the sorted halves into a sorted whole
  }
}

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

```c
int gorenanderDQ(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);
    }
    int maxRight = 0;
    for (int i = m + 1; i <= h; i++) {
        sum += a[i];
        maxRight = max(maxRight, sum);
    }
    int maxL = gorenanderDQ(a, l, m);
    int maxR = gorenanderDQ(a, m + 1, h);
    int maxC = maxLeft + maxRight;
    return max(maxC, max(maxL, maxR));
}
```
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);
    }
}

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
Greedy Algorithms
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, 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.

• 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
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

F(6) = 8

F(5)

F(4)

F(3)

F(2)

F(1)

F(0)

F(3)

F(2)

F(1)

F(0)

F(2)

F(1)

F(0)

F(3)

F(2)

F(1)

F(0)

F(3)

F(2)

F(1)

F(0)
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++)
        f[i] = UNKNOWN;
    return(fib_c(n));
}
Dynamic Programming

F(6) = 8
  /  \
F(5)  F(4)
  /  \
F(4)  F(3)
    /  \
F(3)  F(2)
    /  \
F(2)  F(1)
      /  \
F(1)  F(0)
Dynamic Programming

/* fibonacci by dynamic programming: cache & no recursion */
/* NB: need correct order of evaluation in the recurrence relation */
/* O(1) 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]);
}
Dynamic Programming

/* fibonacci by dynamic programming: minimal cache & no recursion */
/* 0(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);
}
Dynamic Programming

```c
int grenanderDP(int a[], int n) {
    int table[n+1];
    table[0] = 0;
    for (int k = 1; k <= n; k++)
        table[k] = table[k-1] + a[k-1];
    int maxSoFar = 0;
    for (int i = 1; i <= n; i++)
        for (int j = i; j <= n; j++) {
            int thisSum = table[j] - table[i-1];
            if (thisSum > maxSoFar)
                maxSoFar = thisSum;
        }
    return maxSum;
}
```
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

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