CS250 Winter 2003

Week 8, Lecture 3: Fundamental techniques
Bruchetta

Ingredients

• 2 cloves of garlic (finely chopped)

• 3 tbsp olive oil

• Bread (French baguette) sliced about 1/2 inch thick

• 1 tomato (cut in dices)

• Shredded mozzarella or parmesan (to your taste)

• 4 fresh leaves of basil

• 2 tbsp chopped parsley
Directions

- Spread the bread with 2 tbsp of oil and garlic on both sides and put in the oven untill golden.

- Put aside for a while to let them cool down.

- In a bowl mix the tomatoes, basil, parsley and one table spoon of oil.

- Salt and pepper to taste.

- Let it marinate for an hour.

- Spread the mixture on your pieces of bread and cover with the cheese of your choice.

- Broil untill the cheese is melted.
Specialized tools vs general-purpose tools

You could use...

• a bread slicer, a kitchen robot (to dice the tomatoes), and a garlic press, or:

• one good kitchen knife

Expected result...
Four fundamental algorithmic tools

• divide-and-conquer

• amortization

• dynamic programming

• the greedy method
Divide-and-conquer

Lectures 3-2, 3-3, 5-1, 5-2, 5-3, 7-1, and 8-1.

Idea for **algorithms**: divide a problem, recurse on the subproblems, and combine their solutions.

Examples: recursive integer or matrix multiplication, merge sort, insertion sort, etc.

Idea for **data structures**: divide the data, store the sub-data recursively, and combine in an ADT by defining operations on this data.

Examples: recursive lists and trees.
Amortization

Goal: To analyse the performance of data structures.

Idea: Instead of considering the worst-case running time of individual operations, consider the running time of a series of operations.

* The tool of amortization takes into account the interactions between operations.
Overstatement by worst-case analysis

Example: ADT “clearable stack”, has:

- push, pop, top, size, isEmpty (as in week 6, lecture 2)

- clearStack: empties the stack by removing all elements

Look at an array implementation.

Suppose that the clearable stack $S$ has $n$ elements. What is the worst-case running time of a series of $n$ operations?

Answer with operation-by-operation worst-case analysis: $O(n^2)$.

Because the worst case of clearStack() is $O(n)$, and there are $O(n)$ clearStack()’s in the series.
Improved worst-case analysis

**Theorem 1:** Suppose that $S$ is an initially empty clearable stack implemented with an array. Then a series of $n$ operations on $S$ takes time $\mathcal{O}(n)$.

Proof idea:

- If the last clearStack() operation occurred $i$ operations ago, then there could only have been $i - 1$ pushed elements since.

- If $n$ elements are pushed in total, then the cumulative cost of all clearStack()’s is $n$.

- All other operations take time $\mathcal{O}(1)$ each.
Interpretation with amortization

**Definition:** The *amortized running time* of an operation, within a series of operations, is the worst-case running time of this series of operations, divided by the number of operations.

Example: For the clearable stack ADT, the amortized running time of an operation is then $O(n)/n = O(1)$.

Accounting application: assign a fixed price to every clearable stack operation. E.g.: charge 2$ per operation. When an item is pushed, it costs only 1$, so the other $ is saved for the cost of clearing that element by a potential clearStack().
Dynamic programming

- A technique that can take some problems that seem to require exponential time, and produce polynomial-time algorithms for them.

- These algorithms are typically quite simple: a few lines describing some nested loops that fill up a table.
Longuest increasing subsequence

Problem: Given a sequence $x = x_1, x_2, ..., x_n$, what is the length of the longuest common subsequence of $1, 2, ..., n, n + 1, ...$ denoted $y = y_1, ..., y_m$ such that $y_i$ is in $x$ and occurs before $y_i + k$, for any $k$ (but is not necessarily adjacent in $x$).

Example:

```
1, 3, 4, 2, 9, 4, 5, 7
*     *     *     *
```

So the length of the LIS is 5.
LIS with dynamic programming

Consider: for each $j$ such that $1 \leq j \leq n$, what is the length of the LIS of $x_1, x_2, \ldots, x_j$ that includes $x_j$?

Define: $LIS(x, j)$ as the length of the LIS of $x_1, x_2, \ldots, x_j$ s.t. $x_j$ is part of that subsequence.

Then $LIS(x, 1) = 1$, and for $j > 1$:

$$LIS(x, j) = \max_{i : x_i < x_j} \{LIS(x, i)\} + 1$$

Note: In computing $LIS(x, 3)$ and $LIS(x, 4)$, for instance, there are overlapping subproblems. We memorize subproblem results in order not to compute the same one twice.
LIS with DP pseudo-code

Let $x = [x_1, ..., x_n]$, an array of $n$ integers, and let $M = [LIS(1), ..., LIS(n)]$ be an array to store subproblem results.

\[
M[1] = 1
\]

\[
\text{for } j = 2 \text{ to } n \{ \\
\quad m = 0 \\
\quad \text{for } i = 1 \text{ to } j-1 \{ \\
\quad \quad \text{if } x[i] < x[j] \\
\quad \quad \quad m = \max(m, M[i]) \\
\quad \}\n\quad M[j] = m + 1
\}

This algorithm has running time $O(n^2)$.

Other famous application, in bioinformatics: longuest common subsequence (LCS).
The general technique of dynamic programming

The optimization problem to be solved has to involve three components:

1. **Simple subproblems**: There is a way of breaking the global problem into subproblems, which involves only a few indices: $i, j, k$, etc.

2. **Subproblem optimization**: An optimal solution for the global problem necessarily is a composition of optimal subproblem solutions. It is not possible to find a global optimum that is a composition of suboptimal solutions.

3. **Subproblem overlap**: Optimal solutions to unrelated subproblems can contain subproblems in common.
01-Knapsack with dynamic programming

A hiker can carry a knapsack of maximum weight $W$, and can bring some of $n$ items. Each item $i$ has weight $w_i$ and benefit $b_i$.

Problem: to optimize the total “benefit value” of the items brought without exceeding the weight limit $W$.

Naive solution: go through all possibilities, in time $\Theta(2^n)$, by keeping track of the maximum benefit so far, not exceeding the allowed weight.

Dynamic programming solution: how to define the subproblems?
The greedy method

Optimization strategy:

- Sequence of choices to solve an optimization problem.
- Choices start from some well-understood starting condition.
- Iteratively take the following decisions by taking the best choice from all of those that are currently possible.

If this works for problem $P$, it is said to possess the greedy-choice property: the global optimal choice can be reached from local optimal choices.
Fractional knapsack with the greedy method

Suppose the same knapsack problem, but with items that can be arbitrarily broken up.

Greedy strategy: pick the maximum amount of the item with the highest benefit / weight ratio and put it in the knapsack; iteratively do this until nothing more fits.

The fractional knapsack satisfies the greedy-choice property. So “greed is good”.