# Chapter 2

# Lecture 2: Divide and conquer and Dynamic programming

## 2.1 Divide and Conquer

**Idea:**
- divide the problem into subproblems in *linear time*
- solve subproblems recursively
- combine the results in *linear time*, so that the result remains correct.

**Well known example:** Mergesort, sorting $n$ numbers in $n \log n$ time:

- Divide the input into two
- Sort the two halves by recursive MergeSort
- Merge the two sorted list in linear time.

Often Divide and Conquer works for problems were brute force algorithms are polynomial time. In the basic case we divide the problem into two subproblems, but here we start with the general case immediately.

**How can we derive D-C result?** i) We can guess the emerging running time, and prove it by induction, or ii) we can unroll the recurrence, by *analyzing the first levels, identifying the patters, and summing over all levels.*

**Unrolling the recurrence, general approach:** (see figure 4.2 (5.2)) Divide the problem of $q$ subproblems of $n/2$ size, and combine the results in $cn$ time. Note, that the challenge is the division to subproblems, and the combination of the results, such that the result after merging is correct. For running time $T(n)$ we have

$$T(n) \leq qT(n/2) + cn,$$
$$T(2) \leq c.$$

1. Analysing the first few levels:
   First level: $cn$ + all subsequent recursive calls
   Second and third level: $qc(n/2)$ + all subsequent calls, $q^2c(n/2^2)$ + all subsequent calls
2. Identifying the pattern:
   For level i: $(q/2)^i cn$, that is, the algorithm has more and more to do in each layer, if $q > 2$.
3. Summing up over all levels, where we have $\log_2 n$ levels

$$T(n) \leq \sum_{i=0}^{\log_s n - 1} (q/2)^i cn = \frac{(q/2)^{\log_2 n} - 1}{q/2 - 1} cn \leq \frac{c}{q/2-1} n (q/2)^{\log_2 n} \tag{2.1}$$
$$= c_2 n n^{log_2 q - 1} = c_2 n^{log_2 q}. \tag{2.2}$$

where we have used that $a^{\log b} = b^{\log a}$. We can then state the following theorem.

**Theorem 6.** *Any $T(n)$ satisfying $T(n) \leq qT(n/2) + cn$ for $q > 2$ is bounded by $O(n^{\log_2 q})$.*

For example, if you need to divide the problem in $q = 3$ subproblems, the running time will be $O(n^{\log_2 3}) = O(n^{1.59})$. Note, if you have a quadratic brute force approach, then dividing the problem into four subproblems does not help.

**Problem.** Integer multiplication

For $n$ digit numbers, following elementary school methods, the running time is $O(n^2)$. (We multiply one number by one digit at a time, for all $n$ digits.) For simplicity, we consider numbers of same number of digits.

Let us try an alternative method, that leads to Divide and Conquer. We write the numbers as $x_1 2^{n/2} + x_2$, where $x_1$ is the higher order $n/2$ bits, $x_2$ is the lower order $n/2$ bits.

$$xy = (x_1 2^{n/2} + x_2)(y_1 2^{n/2} + y_2) = x_1 y_1 2^n + (x_1 y_0 + x_0 y_1) 2^{n/2} + x_0 y_0.$$

Note, that this leads to a $q = 4$ case, which is unfortunately leads to quadratic running time. We can do it with $q = 3$:

- calculate $x_1 y_1$, $x_0 y_0$
- calculate $(x_1 + x_0)(y_1 + y_0) = x_1 y_1 + x_1 y_0 + x_0 y_1 + x_0 y_0$.
- get the required $x_1 y_0 + x_0 y_1$ from these.

Final algorithm is given in the book, with complexity $O(n^{\log_2 3})$.

Reading: closest points for the difficulties of linear combinations of results. Other D&C problems: counting inversions, Fast Fourier.

Further generalization: similar results when the merging is not in linear time.

## 2.2 Dynamic programming

The name comes from Bellman, it emerged before the wide spread of computers. Dynamic stays changing it time, and programming stays for planning.

Algorithmic paradigms:

**Greedy.** Build up a solution incrementally, by step wise optimization according to some local criterion.

**Divide-and-conquer.** Break up a problem into independent subproblems, solve each subproblem, and combine solutions.

**Dynamic programming.** Break up a problem into a series of overlapping subproblems, and build up solutions to larger and larger subproblems.

### 2.2.1 From recursion to iteration - weighted interval scheduling

#### Recursive solution

**Problem. Weighted interval scheduling.** We have a set $R$ of requests $i$, given by starting and finishing times $(s_i, f_i)$, and a weight $v_i$. $|R| = n$. A subsets of requests is compatible, if no two of them overlap in time. The goal is to find a maximum weight compatible subset.

We do some preprocessing: order requests according to finishing time, and record the last compatible request as $p(i)$.

Recursive thinking: let us consider the optimal solution $\mathcal{O}$, and the last interval. There are two options.

1. $n \in \mathcal{O}$: then $\mathcal{O}$ includes an (the) optimal solution for request $\{1, ...p(n)\}$.

2. $n \notin \mathcal{O}$: then $\mathcal{O}$ is the same as the optimal solutions for requests $\{1, ...n-1\}$.

Based on this we have the following recursive equation:

$$OPT(j) = \max(v_j + OPT(p(j)), OPT(j-1)),$$

and $j \in \mathcal{O}$ iff

$$v_j + OPT(p(j) > OPT(j-1).$$

This gives us a recursive algorithm *Compute-Opt(j)*. See book.

**Theorem 7.** *Compute-Opt(j) correctly computes OPT(j) for each j.*

*Proof.* Proof by induction, and following the same reasoning as above. □

Are we done? No.
Running time of the recursive Compute-Opt: grows according to the Fibonacci numbers, and is therefore exponential.

#### Memorizing the recursion

Compute-Opt becomes exponential, because it calculates the same instance of Compute-Opt(j) for several times. Memorized version, *M-Compute-Opt* stores results once computed. See algorithm in book.

**Theorem 8.** *The running time of M-Compute-Opt(n) is $O(n)$, assuming that the intervals are sorted by their finish times.*

*Proof.* We look for a suitable progress measure that decreases by one each time M-Compute-Opt is called. The number of entries that are empty is such a measure. □

If we also need the optimal set, and not only the maximum weight, then we need additional post processing, Find-Solution(j). See algorithm in book.

**Theorem 9.** *The running time of Find-Solution(n) is $O(n)$.*

*Proof.* This is a recursive solution which calls strictly lower values, that is, completes in $O(n)$ time. □

#### Iterative solution

The key tool for the efficient recursive solution was the maintaining of the array $M$ of already computed values. But then nothing stops us to directly compute the entries of $M$, which leads to an iterative algorithm, *Iterative-Compute-Opt*. See book.

**Theorem 10.** *The running time of Iterative-Compute-Opt(n) is $O(n)$.*

### 2.2.2 Basic Steps of Dynamic Programming

To find a dynamic programming based solution, we need to find a collection of subproblems derived from the original problem that satisfies the following:

1. There are a polynomial number of subproblems. Reasonable requirement to keep the running time polynomial. (We had two for the interval scheduling.)

2. The solution can be easily computed from the solutions to the subproblems. Again, to make sure the algorithm remains polynomial. (It was the maximum operation for interval scheduling.)

3. There is a natural ordering of subproblems from smallest to largest. This is needed to be able to use memorization or iteration. (It was the number of intervals considered, according to increasing finishing times.)

### 2.2.3 Subset sums and Knapsack problems

Here the direct approach of defining subproblems do not work. We demonstrate the technique of adding a new variable.

**Problem. Subset sum problem:** We are given $n$ items with wights $w_i \in \mathbb{N}$, and a bound $W$. Select a subset $S$, such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} w_i$ is as large as possible.

Solution approaches: greedy solutions with starting from the heaviest or lightest item obviously do not work. So we try with dynamic programming. Let us assume we ordered the requests. Unfortunately the recursion as in the interval scheduling does not work, if item $n$ is part of the optimal solution, all the others may be part of it as well, only the limit $W$ is decreased. Since we need to keep track of the remaining space, we introduce one more variable:

$$OPT(i, w) = \max_{S \subseteq \{1 \ldots i\}} \left( \sum_{j \in S} w_j | \sum_{j \in S} w_j \leq w \right)$$

Then we can write up a recursive construction of the optimal set $\mathcal{O}$:

- if $n \notin \mathcal{O}$ then $OPT(n, W) = OPT(n-1, W)$,
- if $n \in \mathcal{O}$ then $OPT(n, W) = w_n + OPT(n-1, W - w_n)$,

or

- if $w < w_i$ then $OPT(i, w) = OPT(i-1, w)$,
- otherwise $OPT(i, w) = \max(OPT(i-1, w), w_i + OPT(i-1, w - w_i))$.

See the algorithm in the book.
For the iterative implementation we need to fill in a matrix $M$ of size $n \times W$. Note, it also requires the assumptions, that weights are integer.

**Theorem 11.** *The Subset-Sum(n, W) algorithm correctly computes the optimal value of the problem and runs in $O(nW)$ time.*

*Proof.* Each entry of $M$ is filled in in $O(1)$ time, by comparing previous entries. □

We call such algorithms pseudo-polynomial, as the running time depends on the largest integer involved in the problem. These algorithms get less practical as this number grows large.

**Theorem 12.** *Given the table $M$, the optimal set $S$ can be backtracked in $O(n)$ time.*

**Problem. Integer Knapsack:** We are given $n$ items with values $v_i \geq 0$, wights $w_i \in \mathbb{N}$, and a bound $W$. Select a subset $S$, such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} v_i$ is as large as possible.

We can use exactly the same approach as for the subset sum problem, with slightly different recursion rule, but the same matrix $M$:

- if $n \notin \mathcal{O}$ then $OPT(n, W) = OPT(n-1, W)$,
- if $n \in \mathcal{O}$ then $OPT(n, W) = v_n + OPT(n-1, W - w_n)$,

or

- if $w < w_i$ then $OPT(i, w) = OPT(i-1, w)$,
- otherwise $OPT(i, w) = \max(OPT(i-1, w), v_i + OPT(i-1, w - w_i)$.

**Theorem 13.** *The Knapsack(n, W) problem can be solved in $O(nW)$ time.*

### 2.2.4 Shortest path in a graph, including negative weights

Recall, the Dijskra algorithm finds shortest path in graphs with positive weights, in $O(m \log n)$ steps.

**Problem. Shortest path with negative weights:** Given $G = (V, E)$, and weights $\{c_{ij}\}$. Decide if $G$ has a negative cycle. If the graphs has no negative cycle, find a path $P$ from $s$ to $t$ with minimum total cost.

We can easily see some wrong approaches. Dijsktra does not work, since now we can not easily declare the set of discovered nodes. Another idea would be to add a large constant to each weights, so that each of them becomes positive. But this penalize long paths more that short paths, and the shortest path may change. Let us see a dynamic programming approach. We will end up with the *Bellman-Ford Algorithm*, which is one of the first applications of dynamic programming. Remember, what we need:

- polynomial number of subproblems
- easy merging of subproblems
- a natural sequence of items to be included.

Moreover, we will use the following simple statement.

**Lemma 9.** *If $G$ does not have negative cycles, then the shortest paths have at most n-1 edges ($|V| = n$).*

We need two tricks from earlier discussions: we will use more than two subproblems, and we will have two variables: $i$ for using at most $i$ edges, and $v$ as the start of a path to $t$ Then we know the following about the optimal path $P$, representing $OPT(i, v)$, that is, shortest path from $v$ to $t$ with maximum $i$ edges:

- if $P$ uses at most $i - 1$ edges, then $OPT(i, v) = OPT(i - 1, v)$,
- if $P$ is uses $i$ edges, and the first edge is $(v, w)$, then $OPT(i, v) = c_{vw} + OPT(i - 1, w)$,

or, as a recursive formula:
if $i > 0$ then

$$OPT(i, v) = \min(OPT(i - 1, v), \min_w(OPT(i - 1, w) + c_{vw})).$$

The iterative shortest path algorithm fills in an $M$ matrix of $n \times n$. To fill in an entry, it needs to compare $n$ previous entries. See algorithm in book.

**Theorem 14.** *The Shortest-Path algorithm correctly computes the minimum cost of an s-t path in a graph with no negative cycles, and runs in $O(n^3)$ time.*

*Remark.* The algorithm can be implemented in $O(mn)$ time, where $n^2$ is changed to $m$ since only $w - v$ pairs with edges need to be evaluated. Similarly, the space requirements can be decreased to $O(n)$.

This shortest-path algorithm is relevant even for graphs with only positive weights, due to the possibility of distributed implementation. In the distributed implementation each node pushes its new shortest path values to its neighbors, neighbors update their own values, etc. Algorithms with synchronous and asynchronous implementation in the book. The synchronous distributed implementation is the official Bellman-Ford algorithm.

### 2.2.5 Negative Cycles in Graphs

Actually, up to now we solved only half of the shortest path problem. We have not discussed the detection of cycles yet. Note two subcases: the negative cycle is on a path to the sink, or the negative cycle is not on a path to the sink. If we want to detect all cycles, then we need to use an augmented graph, adding a node $t$ and an edge from all nodes to $t$.

**Lemma 10.** *The augmented graph $G'$ has negative cycle such that there is a path from the cycle to the sink $t$ iff $G$ has negative cycle.*

*Proof.* Forward direction (If $G'$ has negative cycle, then $G$ has too): Since no edge leaves $t$, the negative cycle can not go through $t$, and needs to be there in $G$ as well.

Backward direction (If $G$ has negative cycle, then $G'$ has too): If $G$ has a negative cycle, it will not disappear due to the augmentation. Also, a path needs to go to $t$, since there is an edge from all nodes to $t$. □

**Algorithm.** Bellman-Ford extension to detect cycles.

Take B-F, and let it run for $i \geq n$.

**Lemma 11.** *If a node $v$ can reach $t$ and it is contained in a negative cycle, then*

$$\lim_{i \to \infty} OPT(i, v) = -\infty.$$

**Lemma 12.** *If $G$ has no negative cycles, then $OPT(i, v) = OPT(n-1, v)$, $\forall v$, $\forall i \geq n$.*

The question is for how large $i$ do we need to test? e.g., $OPT(n, v) = OPT(n-1, v)$ may hold for a $v$ in a negative cycle (an entire loop needs to be made, so maybe there is no path which is just one link longer). Therefore, the following result is important.

**Theorem 15.** *There is no negative cycle with path to $t$ iff $OPT(n, v) = OPT(n-1, v)$ holds $\forall v$.*

*Proof.* If no negative path then $OPT(n, v) = OPT(n-1, v)$ holds: see previous lemma.

If $OPT(n, v) = OPT(n-1, v)$ then no negative path: OPT values for iteration $i$ are calculated only from the values in iteration $i-1$, so if they have not changed in one iteration, they will never change. This means there can not be a negative cycle, since then the lemma before the last one would hold. □

**Corollary 2.** *If in a graph $G$ with $n$ nodes $OPT(n, v) \neq OPT(n-1, v)$, then there is a path from $v$ to $t$ containing a negative cycle. This also helps to backtrack the negative cycle in $O(n)$ steps.*

### 2.2.6 Other dynamic programming problems

- Segmented least squares: example for multi-way choices,
- Sequence alignment
- Parsing

## 2.3 What have we learned

Divide and Conquer:

- Solve subproblems $q$, and combine results in linear time
- Running time $O(n^{\log_2 q})$
- Examples: sorting, multiplication, closest points, convolution

Dynamic programming:

- Similarly to divide and conquer we build up solution from subproblems
- Design steps: recursion, recursion with memorization, iterative solution
- To derive complexity: see how entries in the iterative solution fill up
- Famous problems: weighted interval, subset sum, knapsack, shortest paths