General Training for Programming Competitions / Coding Interviews

Backtracking, Divide-and-Conquer, Dynamic Programming, and Greedy Algorithms

Andreas Goral

Friedrich Schiller University Jena

Winter Semester 2022/2023

What's The Lecture About?

In this lecture we cover some **strategies** for **algorithm design**.

- 👣 Handle repetitive tasks through **iteration**,
- 💃 Iterate elegantly using **recursion**,
- 💪 Use **brute force** when you're lazy but powerful,

- 🏃 Test bad options and then **backtrack**,
- ⚔️ **Divide and conquer** your toughest opponents,
- 💉 Identify old issues **dynamically** not to waste energy again,
- ⏳ Save time with **heuristics** for a reasonable way out.

The **iterative strategy** consists in using loops (e.g. for, while) to repeat a process until a condition is met. (merge two sorted arrays)

The **recursive strategy** comes to mind for solving a problem defined in terms of itself. (traversing a binary tree)

The **brute force strategy** solves problems by **inspecting all** of the problem's possible **solution candidates**. (= exhaustive search)

Backtracking → Rücksetzverfahren

Is a **general algorithm / technique** for **finding all** (**or some**) **solutions** to some **computational problem**, that **incrementally builds candidates** to the solutions, and **abandons** each **partial candidate *c*** ("backtracks") **as soon as it determines** that ***c* cannot be completed to a valid solution**.

Basically, we carry out an **<u>exhaustive</u>** ***depth-first search***, therefore **we need** in backtracking **effective pruning techniques**. (backtracking problems have **exponential / factorial runtimes**)

The **general algorithm** works as follows:

At each step in the backtracking algorithm, we start from a given partial solution, say, $a = (a_1, a_2, \ldots, a_k)$, and try to extend it by adding another element at the end. After extending it, we must test whether what we have so far is a solution, if so, we should print it, count it, or do what we want with it. If not, we must then check whether the partial solution is still potentially extendible to some complete solution. If so, recur and continue. If not, we delete the last element from a and try another possibility for that position, if one exists.

Basic *Slow* Template For Backtracking

backtrack_template.cpp

```
bool is_solution(const vector<int> &a, size_t n) {
  // This Boolean function tests whether the elements in the vector are a
  // complete solution for the given problem.
  return false;
}

void process_solution(const vector<int> &a, void *out) {
  // This routine prints, counts, or somehow processes
  // a complete solution once it is constructed.
}

vector<int> construct_candidates(const vector<int> &a, size_t n) {
  vector<int> candidates;
  // Compute here possible candidates for the next position of a.
  return candidates;
}

// "n" is problem size; with "finished" you can kill recursion;
// with the "out" pointer you can get results out from "process_solution"
void backtrack(vector<int> &a, int n, bool &finished, void *out) {
  if (is_solution(a, n)) { // if we found a solution, process it
    process_solution(a, out);
  } else {
    vector<int> candidates = construct_candidates(a, n);
    for (const int candidate : candidates) {
      a.push_back(candidate);          // add candidate to possible solution
      backtrack(a, n, finished, out); // recurse
      a.pop_back();                    // remove candidate from solution
      if (finished) { return; }        // terminate early
    }}}
```

Constructing All Subsets With Backtracking

We can construct the 2^n subsets of n items by iterating through all possible 2^n length-n vectors of true or false, letting the ith element denote whether item i is or is not in the subset.

subsets.cpp

```cpp
bool is_solution(const vector<int> &a, size_t n) {
  return a.size() == n; // test if solution size reached
}

void process_solution(const vector<int> &a, void *out) {
  for (size_t i = 0; i < a.size(); ++i) {
    if (a[i]) // if item i is in subset, print it
      cout << i << " ";
  }
  cout << endl;
}

// It's a rather inefficient solution!
vector<int> construct_candidates(const vector<int> &a, size_t n) {
  return {0, 1}; // 0 means is not subset, 1 means is in subset
}
```

This is not about efficiency! Of course, we could represent a subset as a single integer for moderate problem sizes.

Constructing All Permutations With Backtracking

Print with backtracking all permutations of size n. For example, if $n = 2$ the valid permutations are $< 0, 1 >$ and $< 1, 0 >$. There are $n!$ permutations.

permutations.cpp

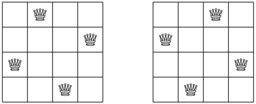
```cpp
bool is_solution(const vector<int> &a, int n) {
  return int(a.size()) == n; // test if solution size reached
}

void process_solution(const vector<int> &a, void *out) {
  for (auto number : a)
    cout << number << " "; // print permutation
  cout << endl;
}

// It's a rather inefficient solution!
vector<int> construct_candidates(const vector<int> &a, int n) {
  vector<int> candidates;
  vector<bool> in_permutation(n, false); // vector of size n, all entries false
  for (auto number : a)
    in_permutation[number] = true; // flag already used numbers
  for (int i = 0; i < n; ++i)
    if (in_permutation[i] == false) // we prune away invalid candidates
      candidates.push_back(i);
  return candidates;
}
```

The n-Queens Problem

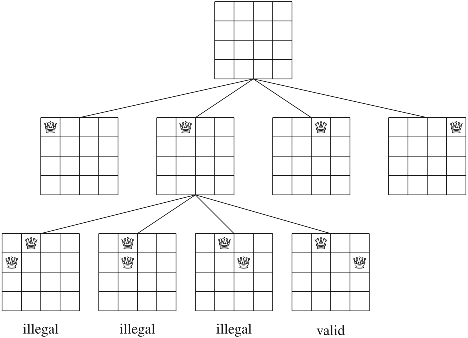
Compute the **number of ways** n **queens can be placed** on an $n \times n$ **chessboard** so that **no two queens attack each other**.

The n-Queens Problem

Compute the **number of ways n queens can be placed** on an $n \times n$ **chessboard** so that **no two queens attack each other**.



The possible ways to place 4 queens on a 4 × 4 chessboard



Partial solutions to the queen problem using backtracking

The n-Queens Problem Code

n_queens.cpp

```
bool is_solution(const vector<int> &a, int n) {
  // vector a is always valid, so is_solution check is trivial
  return int(a.size()) == n; // test if solution size reached
}

void process_solution(const vector<int> &a, void *out) {
  size_t &counter = *((size_t *)out); // make integer reference from void pointer
  counter++; // increment counter, since we found a solution
}

vector<int> construct_candidates(const vector<int> &a, int n) {
  // ith element of the vector "a" lists the column
  // where the ith (row) queen resides --> possible column candidates [0, n - 1]
  vector<int> candidates;
  // pruning idea: no two queens may lie on the same row, column or diagonal
  // since in vector "a" each position represents a different row, we only
  // need to ensure that on the columns and diagonals there are no conflicts
  for (int i = 0; i < n; i++) {
    bool legal_move = true;
    for (int j = 0; j < int(a.size()); j++) {
      if (int(a.size()) - j == abs(i - a[j])) // diagonal threat
        legal_move = false;
      else if (i == a[j])   // column threat
        legal_move = false; // another queen occupies the column already
    }
    if (legal_move)
      candidates.push_back(i);
  }
  return candidates;
}
```

Divide and Conquer

A **Divide and Conquer** strategy solves a problem by **dividing** it **into two** (sometimes more) **independent identical sub-problems**, **ideally each** about **half the size of the original problem**, and **solving** the **sub-problems recursively**.

The strategy consists in **three steps** applied at **each level of recursion**:

- a **divide phase** in which the problem is subdivided into a number of smaller and easier to solve sub-problems
- a **conquer phase** where the sub-problems are solved recursively, while simple problems (not large enough) can be solved directly without further recursion
- a **combine step** in which the solutions of the sub-problems are merged to obtain the solution of a bigger problem

Quicksort: **divide** (partitioning with pivot); **conquer** (recursive calls to sort small sub-array); **combine** (absent)

For the **most Divide and Conquer approaches** one of the **two** following **patterns** apply (somethimes it's a mix $\rightarrow$ Strassen algorithm for matrix multiplication):

- **divide phase** with **negligible computation cost**, **most** of the **running time** is spent in the **combine phase** (for example merge of two sorted sub-arrays in mergesort)
- **combine phase** is **totally absent** and the **entire work** is essentially performed in the **divide phase** (partitioning in quicksort)

$\rightarrow$ To be fast, we have to parallelize also the work intensive part of Divide and Conquer

Parallelization friendly Divide and Conquer

The **Holy Grail** of **Divide and Conquer in terms** of **parallelization** are approaches where **divide** and **combine phases** are **cheap** (like $\mathcal{O}(\log n)$) or absent. $\rightarrow$ **Multithreaded 2 Way Merge with Divide and Conquer**

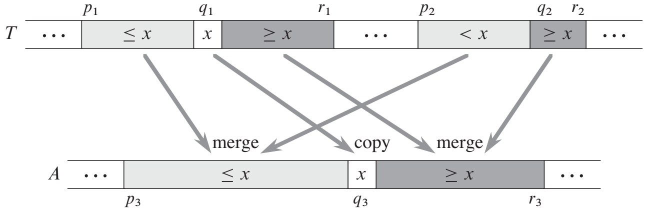


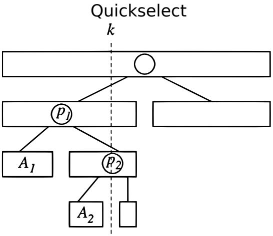
Figure 27.6 The idea behind the multithreaded merging of two sorted subarrays $T[p_1 \mathinner{..} r_1]$ and $T[p_2 \mathinner{..} r_2]$ into the subarray $A[p_3 \mathinner{..} r_3]$. Letting $x = T[q_1]$ be the median of $T[p_1 \mathinner{..} r_1]$ and q_2 be the place in $T[p_2 \mathinner{..} r_2]$ such that x would fall between $T[q_2 - 1]$ and $T[q_2]$, every element in subarrays $T[p_1 \mathinner{..} q_1 - 1]$ and $T[p_2 \mathinner{..} q_2 - 1]$ (lightly shaded) is less than or equal to x, and every element in the subarrays $T[q_1 + 1 \mathinner{..} r_1]$ and $T[q_2 + 1 \mathinner{..} r_2]$ (heavily shaded) is at least x. To merge, we compute the index q_3 where x belongs in $A[p_3 \mathinner{..} r_3]$, copy x into $A[q_3]$, and then recursively merge $T[p_1 \mathinner{..} q_1 - 1]$ with $T[p_2 \mathinner{..} q_2 - 1]$ into $A[p_3 \mathinner{..} q_3 - 1]$ and $T[q_1 + 1 \mathinner{..} r_1]$ with $T[q_2 \mathinner{..} r_2]$ into $A[q_3 + 1 \mathinner{..} r_3]$.

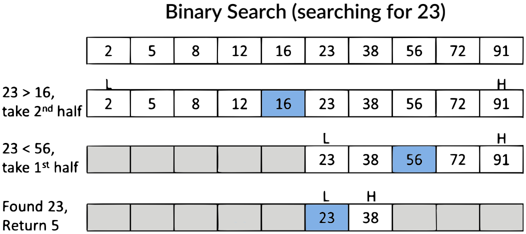
Introduction to Algorithms, 3rd edition, p. 798.

Pseudo Divide and Conquer

There is a class of algorithms called Divide and Conquer, but **actually they are not**. They do not split the problem into two (or more) similar sub-problems, but successively break the original problem down into **one smaller sub-problem**.
(We have **only one conquer call** in each recursive level.)

→ The name **Decrease and Conquer** has been proposed for the single-sub-problem class of "Divide and Conquer" algorithms.





Dynamic Programming

Like *Divide and Conquer*, ***Dynamic Programming*** **solves the problem by combining the solutions of multiple smaller problems**, but what makes *Dynamic Programming* different is that the **same sub-problem may reoccur**. Therefore, a **key** to making *Dynamic Programming* efficient is **caching the results of intermediate computations**.

We need to find a way to **break the original problem into sub-problems** such that we can solve the original problem relatively easily once cached solutions to the sub-problems are available. Dynamic programs are mostly implemented by **recursion**. Alternatively, **you can study the pattern** of the **recursive calls** and implement them **iteratively**. You still "cache" previous work. **Minimizing cache space** is a **recurring theme** in Dynamic Programming.

Dynamic Programming

Dynamic programs can be solved in **bottom-up** (iterative, easier to make the cache small) or **top-down** fashion (recursive, sometimes easier to implement, possibility of pruning). $\rightarrow$ For example, think about the different ways to implement the computation of the nth Fibonacci number.

Some people call **top-down** Dynamic Programming ***memoization*** and only use Dynamic Programming to refer to bottom-up work.

Fibonacci Numbers With Dynamic Programming

fibonacci.cpp

```cpp
// recursive implementation, O(1.618^n) time, 1.618 --> golden ratio
uint64_t fibonacci(uint64_t n) {
  if (n <= 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
}

// top-down dynamic programming (or memoization), with recursion, O(n) time
uint64_t recurse(uint64_t n, vector<uint64_t> &memo) {
  if (n <= 1) return n;
  if (memo[n] == 0) memo[n] = recurse(n - 1, memo) + recurse(n - 2, memo);
  return memo[n]; // lookup number
}
uint64_t fibonacci_top_down(uint64_t n) { // call this function
  vector<uint64_t> memo(n + 1); // create cache for intermediate results
  return recurse(n, memo);
}

// bottom-up dynamic programming, no recursion, O(n) time
uint64_t fibonacci_bottom_up(uint64_t n) {
  if (n <= 1) return n;
  vector<uint64_t> memo(n); memo[1] = 1; // create cache for intermediate results
  for (uint64_t i = 2; i < n; i++) memo[i] = memo[i - 1] + memo[i - 2]; // compute missing number
  return memo[n - 1] + memo[n - 2];
}

// bottom-up dynamic programming, O(n) time, small cache
uint64_t fibonacci_bottom_up_small_cache(uint64_t n) {
  if (n == 0) return n;
  uint64_t a = 0, b = 1; // reduce space complexity of the cache (we only need the last two numbers)
  for (uint64_t i = 2; i < n; i++) { uint64_t c = a + b; a = b; b = c; }
  return a + b;
}
```

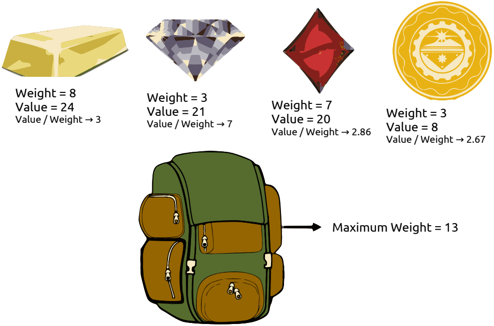
Heuristics $\rightarrow$ in Our Case It's Being Greedy

A **heuristic**, is a method that **leads** to a **solution without guaranteeing it's** the **best** or **optimal** one (actually, there are some heuristics that can guarantee optimality). Heuristics can help when methods like brute force or backtracking are too slow. Heuristics can also be used within other methods (like backtracking) to speed up the computation. There are many funky heuristic approaches, but **we'll focus on** the simplest: **greedy**.

The greedy approach consists in **never coming back to previous choices**. It's the **opposite of backtracking**. Try to make the **best choice at each step**, and don't question it later. $\rightarrow$ In other words: At each step the algorithm makes a **decision** that is **locally optimum**, and it **never changes that decision**.

A Greedy Knapsack Heuristic

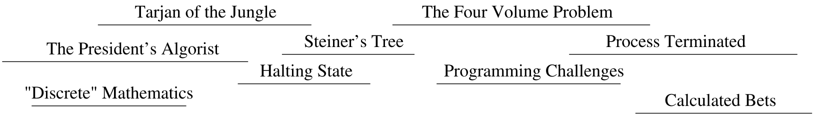
A greedy burglar breaks into a museum to steal some valuable objects. He has a knapsack that can carry a weight of up to 13 (we ignore units). Which items will he steal? Remember, the less time he spends in the museum, the less likely he gets caught.



A greedy packer will probably put the highest valued items in the knapsack until he can't fit more. Unfortunately, there is no optimal heuristic for the "general" 0-1 Knapsack Problem $\rightarrow$ If the weights have arbitrary precision, it's NP-hard $\rightarrow \mathcal{O}(2^n)$. But if the weights are positive integers you can solve the 0-1 Knapsack Problem with dynamic programming in $\mathcal{O}(nw)$ time, where n is the number of objects and w the weight limit of the knapsack.

Non-Overlapping Movie Scheduling Problem

Problem: Movie Scheduling Problem
Input: A set I of n intervals on the line.
Output: What is the largest subset of mutually non-overlapping intervals which can be selected from I?

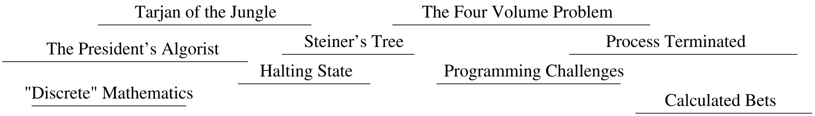


An instance of the non-overlapping movie scheduling problem

Let's try some greedy heuristics:

Non-Overlapping Movie Scheduling Problem

Problem: Movie Scheduling Problem
Input: A set I of n intervals on the line.
Output: What is the largest subset of mutually non-overlapping intervals which can be selected from I?



An instance of the non-overlapping movie scheduling problem

Let's try some greedy heuristics:

EarliestJobFirst(I)

  Accept the earlest starting job j from I which does not overlap any previously accepted job, and repeat until no more such jobs remain.

ShortestJobFirst(I)

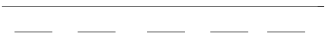
  While $(I \neq \emptyset)$ do

    Accept the shortest possible job j from I.

    Delete j, and any interval which intersects j from I.

The Algorithm Design Manual, 2nd edition, pp. 9–11.

A Greedy Algorithm That Is Optimal



(l)

(r)

Bad instances for the (l) earliest job first and (r) shortest job first heuristics.

At this point, an algorithm where we try all possibilities may start to look good:

ExhaustiveScheduling(I)
  $j = 0$
  $S_{max} = \emptyset$
  For each of the 2^n subsets S_i of intervals I
    If (S_i is mutually non-overlapping) and ($size(S_i) > j$)
      then $j = size(S_i)$ and $S_{max} = S_i$.
  Return S_{max}

But wait a minute, what about this heuristic?

OptimalScheduling(I)
  While ($I \neq \emptyset$) do
    Accept the job j from I with the earliest completion date.
    Delete j, and any interval which intersects j from I.

Exercises (Upload Solutions to your Repository)

1. Solve the **15-puzzle** with **backtracking**.

   fifteen_puzzle.cpp

2. **Read** about **dynamic programming** and **implement** **one example iteratively**, and, **with recursion**.

   dynamic_programming.pdf   dynamic.cpp

3. **Compute** the **duration** in an **optimal task assignment**.

   optimal_duration.cpp

Exam Questions

- **What is backtracking**?
- **How** would you proceed to **compute** the ***n*-Queens** problem **with backtracking as fast as possible**?
- Describe some **ideas how** to **solve the 15-puzzle** with **backtracking**. (Tell me what you think is important.)
- **Why** is **efficient parallelization not trivial** in **most divide and conquer algorithms**.
- Why may ***decrease and conquer*** be a **better name choice** for algorithms like **quickselect** or **recursive binary search than** calling them ***divide and conquer***?
- **What is dynamic programming**?
- **Compare** the **top-down approach** with the **bottom-up approach** used in **dynamic programming**.
- **Present briefly** a **greedy heuristic** of **your choice**.