0.1 Mission Statement

This is our personal collection of some important ideas in computer science and programming. It shall help us document high-level insights that can serve us in practice. We do not try to capture topics in-depth, but instead just try to make us aware, so that we know what to look for when we are in need. We are aware that when everything is important, nothing is important.

We also try to document some of our occasional nuggets of temporary insights.

0.2 Main Topics

- Nothing substantial yet

0.3 Sources to review

- Study notes
  - Compiler Construction
  - Experimental Economics: Design Elements (e.g., baseline neighborhood) and the basic idea of falsification as supplement material for the algorithm engineering notes
  - Formal Systems (something in there?)
  - ~~Advanced Data Structures~~
  - Parallel Algorithms
  - Parallel Machines and Parallel Programming
  - Randomized Algorithms
  - Computerarchitecture (implications of branch predictions, cache coherence, pipelining, super scalar architecutures ...)
  - Softwareengineering II
  - Game Theory (Battle of the sexes :p)
- Lecture notes and books to skim
  - Algorithm Engineering
  - Algorithm II
  - Algorithms and Data Structures — The Basic Toolbox
  - Linder's *Things Thy Should have taught you*
- Books with potential:
  - Pragmatic Thinking and Learning
  - Notes from the Pragmatic Programmer

– Notes from Head First Software Development (Very light stuff but maybe there is something in there. I used to enjoy reading it.)

– ...

1 Desigining Abstractions

- Single Level of Abstraction
- Single Responsibility Principle
- Separation of Concerns
- Interface Segragation Principle
- Liskov Substitution Principle
- Open Closed Principle
- Law of Demeter

If I am not totally mistaken, these principles are well described within SICP.

2 Algorithm & Data Structures Design Toolbox

2.1 Data Structures

Many applications just rely on dynamic list structures, hashmaps and sorting. This is fine in most cases. However, here are some observations:

- *Orderness* seems to be underutilized. Instead of storing elements without a particular order, consider keeping them sorted. This can help to speed up operations on them. So for example, instead of hashing and sorting, also consider an ordered associative array (e.g., Java's `SortedMap` or `NaviagableMap`).
- Integer keys seem to be underutilized. This also implies that hashing is favored in cases where plain arrays or bit sets (e.g., Java's `BitSet`) would do just fine (and would even be faster and require less space). Therefore, consider enumerating the elements you are dealing with.
- Using the implementation of an abstract data type does not relieve from having a good mental model of its associate costs. For example, when using a resizable array implementation (e.g. Java's `ArrayList`):

- Try to avoid inserts/deletes at all positions beside the end of the list, as these require the array to be shifted.
- When the number of elements that the list shall hold is known, but elements can only be inserted one by one, correctly initialize the list in order to avoid the numerous internal array copies.

2.1.1 Trie

A *trie* is a special kind of edge labeled tree. If used over a collection of keys, it can be used to find the significant difference between these keys (i.e., the *important bit positions* sufficient to distinguish all keys). For sorted sequences it can be constructed in $\mathcal{O}(n)$. Examples:

- *Fusion Trees* compress keys within *B-Tree* nodes by reducing them to the bits at the important bit positions. All keys of a node can then be fused into a single machine word and compared to a query element in parallel using a single bitparallel subtraction.
- *String B-Trees* use *Patricia Tries* to identify the subset of characters relevant for the comparison of a string pattern to the keys of a *B-Tree*. Knowing these characters and the lengths of the common prefixes of keys, the number of required I/Os per traversal can be limited.
- *Signature Sort* splits keys into chunks and compresses these chunks using a hash function. A trie is then used to filter the chunks that are not relevant for the sort order of the keys. The keys therefore become smaller and sorting them becomes easier.

2.1.2 Suffix Arrays and Suffix Trees

TODO once studied.

2.2 Recurring Design Ideas

There seem to be some simple ideas that have influenced the design of many algorithms and data structures. This list is not authoritative (in particular w.r.t. to naming), it just lists some observations. The ideas seem to be heavily connected and seldomly used in isolation.

- **Tradeoff:** Inspect simple, naive solutions with contradicting space and runtime bounds (e.g, online computation vs. precomputation of all results) and try to find a tradeoff that achieves the best of both worlds. So for example, instead of precoumputing all results, precompute just a particular subset. Examples:

  - Level Ancestor Queries (to find the tree-ancestor on a particular level) can be approached using *jump pointers* and *ladder decomposition*, which both precompute a subset of all potential queries and run in $\mathcal{O}(\log n)$ time. However, a combination of both leads to constant query time and a linear space requirement, thus heavily improving over the computation of all results.

- **Combination:** Combine several different data structures so that expensive operations of one data structure can be improved using a specialized data structure for this particular sub-problem. Examples:
  - *String B-Trees* build *Patricia Tries* on top of individual B-Tree nodes to improve over the binary search to find the insertion point within the keys of a node.

- **Indirection/Bucketing:** Instead of working on a large problem set (with larger accompanying space & runtime bounds), first find a suitable bucket and then solve the problem only within this particular bucket. Examples:
  - *Perfect Hashing* uses an indirection so that it is not required to find a perfect (injective) hash function for all elements, but only for the elements within the same collision bucket. This reduces the overall space requirement while retaining the expected construction and query time.
  - *Y-Fast Tries* improve upon the space requirements of *X-Fast Tries* by building the trie over representatives of buckets instead of over all elements. The buckets are implemented as balanced binary trees and do not just reduce the space requirement of *X-Fast Tries* but also help to reduce update costs using amortization.
  - *Succinct Data Structures* can profit from indirection layers as they reduce the number of elements that have to be distinguished from the whole universe down to all elements within a bucket. Thus, to distinguish these elements less bits are required.

- **Decomposition:** Instead of dealing with elements of objects as a whole, split them into logical subparts. Exploit this inner structure to achieve higher efficiency and eventually solve the problem within a richer model of computation (e.g., compare bits instead of whole integers) . Examples:
  - *X-Fast Tries* split integer keys into their common prefixes and use a binary search to find the longest common prefix.
  - *van Emde Boas Trees* split recursively split keys into a top and bottom halve in order to efficiently navigate within buckets and summaries over these buckets.
  - Level Ancestor Queries can be implemented via the idea of recursively splitting a tree into its longests paths (*longest path decomposition*) and precomputing the results within these paths.

- **Input Reduction/Simplification:** Reduce the problem set at hand until it is simple enough so that it can be solved using basic means. This idea is at the core of most recursive algorithms.
  - *Independet-Set Removal* is used to speed up list ranking.
  - *Signature Sort* recursively reduces the size of the keys until they can be sorted with *Packed Sorting*.
  - The *findclose* operation on *Succinct Trees* finds the position of a closing bracket within the bitstring representing the tree. The idea is to solve the problem for a corresponding, so-called pioneer and then deduce the actual closing position from the closing position of the corresponding pioneer.
- **Broadword Computing:** Fuse data elements into machine words and run operation on then in parallel, instead of sequentially. You can get more done within fewer instructions and without having to hit memory.
  - *Fusion Trees* compare several integer keys with a single bitparallel computation.
  - *Packed Sorting* is a variant of mergesort that packs several short integer keys into a machine word. It then uses bitparallel computations to speed up the base case and the merge of sorted words. In particular, it relies on a bitparallel version of *Bitonic Sorting*.
- **Two Phase Computation:** If something is complicated (and thus slow) to do in a single pass, feel free to use several phases. Examples:
  - *Super Scalar Sample Sort* sorts in two phases. In the first pass it is only decides in which bucket an element shall be assigned to. The actual data movement into the preallocated subarrays is then performed in the second pass once the total bucket sizes are known.
  - Pipelined Prefix Sum (Paralag und Sanders Paper?)
  - All to all for irregular message sizes

2.3 Succinct Data Structures

Succinct Data Structures are space efficient implementations of abstract data types (e.g., trees, bit sets) that still allow for efficient queries.

2.4 Randomized Algorithms

- Among others, randomized algorithms help to improve robustness concerning worst-case inputs (e.g., think of the pivot selection problem in `quicksort` when the sequence is provided by a malicious adversary).

- Allowing randomized algorithms to compute a *wrong* result (with a very low probability) can open many new possibilities concerning speed, space, quality and ease of implementation (e.g., think of *bloom filters* or *Approximate Distance Oracles*).
- Never use the `C rand()` function. If in doubt, use *Mersenne Twister*.
- Use random numbers with care. Treat them as a scarce resource.
- In certain parallel setups, *expected* bounds of randomized algorithms do no longer hold: Consider `n` processes that call operations with *expected* runtime bounds and that have to be synchronized before and afterwards. The runtime will suffer whenever at least one of the processes hits an expensive case.

2.5 Parallel Algorithms

Goal is to come up with a parallel algorithm that has an appropriate computation / communication ratio for the target architecture (e.g., more coarse grained for NORMA or NUMA systems than for SMP systems).

A rough roadmap (see Foster):

1. Start with the problem. Not with an particular algorithm.
2. Partition the problem (Use *Domain* Decomposition or *Functional* Decomposition. See below. You can even combine both recursively.). Expose as much parallelism as possible. Find all the concurrent tasks.
3. Understand the communication paths and required data flow.
4. Aim to reduce communication costs by merging some of the fine-grained tasks that have too many data-dependencies or by duplicating some of the computations whose results are required by many different tasks.
5. Map the resulting tasks to the target architecture and its available PEs. Aim to reduce the overall execution time by allocating the tasks wisely. If required, add a load-balancing mechanism.

- Obviously, PEs should never idle. This implies several things:
  - Tree-shaped communication paths and pipelining can help to distribute data more quickly, so that all PEs can start working earlier.
  - Expose the parallelization from the beginning (e.g., don't spawn threads for the multiple recursive calls within a recursive algorithms such as quicksort)

2.5.1 Functional Decomposition

Inspect the different computation steps. Group them to components that can operate in parallel. Finally, inspect the data required by the different components.

Functional decomposition yields modular program but which has often a limited scalability.

Examples: A pipeline is a special kind of functional decomposition.

2.5.2 Data Parallelism / Domain Decomposition

A partitioning of the data (applies to input, output or heavily used data structures) with a parallel computation on the different partitions.

- Start with a simple PRAM algorithms and take as many PEs as you want to and place them in an appropriate layout (e.g., hypercube, 3D-cube, 2D-grid). This allows you to learn about the problem and how it can be parallelized. Commonly, the next step is then to apply *Brent's Principle* to make the algorithm more efficient by reducing the number of PEs. This can be done by decoupling the strict mapping of elements to PEs and assigning $\frac{n}{p}$ elements per PE for a now arbitrary p.
- Long paths (e.g., linked-lists, long paths within trees) are difficult to parallelize. Sorting the nodes of a path (e.g., via *list ranking* based on *doubling* and *independent set removal*) and storing them in an array, enables parallelization. Every PE can then operate on a particular index-range on the array.
- Load balancing is important and therefore at the core of many parallel algorithms. Some examples:
  - Sample sort uses samples to find splitters that split up the data in (hopefully) equally sized chunks. Each PE is then responsible to sort one of these chunks.
  - Prefix-sums / scan operations are commonly used to enumerate items on the different PEs. Knowing how many of these items exist in total and on each predecessor PE, the items can be distributed equally.

2.6 Algorithm Engineering

Performance depends on technological properties of machines like parallelism, data dependencies and memory hierarchies. These play a role even in lower order terms.

Always run your experiments on:

- different architectures (e.g., current Intel, AMD, MIPS to cover both CISC and RISC architectures)
- different types of inputs (e.g., uniformly distributed, skewed)

3 Computer Architecture

3.1 Numbers and Measurements

- The *absolut* speedup over the best sequential competitor is what counts, not the arbitrary *relative* speedup.
- Watch out for sequential portions of the code. They may heavily limit the overall speedup, as can be shown by Amdahl's Law: $T(n) = T(1)\frac{1-a}{n} + aT(1)$ with a being the fraction non-parallelized portion of the code. For $n \to \text{inf}$ observe that the speedup is bound by $S(n) = \frac{1}{a}$.

3.2 Layers of Parallelization

Often one has to deal with more than one layer of parallelization. The common ones:

- Interconnection of Computers (distributed systems such as clusters or grids): Parallelization of mostly independent tasks.
- Processor Coupling (Message Passing, Shared Memory, Distributed Shared Memory): Explicit communication between and synchronization of tasks.
- Processor Architecture (Pipeling, Superscalar, Multi/Hyperthreading): Parallelization of a sequential instruction stream (via redestribution & overlapping, e.g., dynamic out of order execution by a superscalar unit) or overlapping of different instruction streams (e.g., IOs triggered by cache-faults of one thread, overlapped with the computations of another thread via hyperthreading) .
- SIMD: Instructions broken up into sub-instructions that can be executed in parallel (on different data elements).

3.3 Random Tips

- If the compiler is bad, consider loop unrolling / software pipelining to expose the concurrent instructions to the CPU so that they can be executed in parallel.

- For simplificiation, just assume that the branch-predictor will predict the branch that was taken *the last time*. When designing algorithms, try to void hard-to-predict branches (e.g., the comparison-if in a simple *quicksort*). If possible, convert these convert these control-dependencies to data-dependencies. Using predicated instructions these are then easier/faster to execute because there is no need to flush the pipeline. Examples:
  - Super Scalar Sample Sort with the algorithmic insight that element comparisons can be decoupled from expensive conditional branching.
- Prevent *false-sharing* (i.e., several threads operate on different data elements that happen to reside within the same cache-line). Otherwise, the cache coherence protocol (e.g., MESI, MOESI) will make your CPUs a hard time by constantly invalidating the shared cache-line upon write operations.
- Your language probably has a *weak consistency* assumption. Without proper synchronisation there is no way to predict when or in which order threads will see values written by other threads. Different threads will therefore probably see *different values*. Use the built-in synchronization primitives to prevent this problem.

4 Personal Development

- Always life on the edge of incompetence. When ever you have the chance, approach challenges beyond your current skill level. Be focused, determined and passioned and you will be fine, because even if you fail (what you won't), you have learned a great deal.
- Theres no speed limit in life. You don't have to move (improve, learn, ...) as slowly and over-cautiously as others.
- Never presume to already know. Have a beginner's mind.
- *"Money is nothing more than neutral proof that you're adding value to people's lives. Making sure you're making money is just a way of making sure you're doing something of value to others."* – Derek Sivers

4.1 Studying

- Approach study tasks like you approach programming tasks (see the common principles)
- Do not expect teachers to teach you (they only present information). Nobody will teach you anything. You have to teach yourself.

- Studying is not about time, it's about effort. You have to approach a difficult topics from different angles and discuss it with peers. Make the ideas your own by discovering them for yourself (e.g., with your own examples). Stupid repetition just does not cut it. If you have to memorize something, you haven't understood it.
- Don't miss the big picture by getting lost in all the details. Always zoom-out again and fish for connections and insights.
- Always write down key insights as succinctly and precisely as possible. Feel the pain. You have to do it, because it is a different thing to believe to understand something and to be able to articulate it.
- If stuck with anything, ask yourself *what exactly* is it, that you don't understand? (See the notes on debugging).

4.2 Stress & Burnout Prehab

- Stress kills creativity and gives you a bad hangover. Don't get back to work just because you are bored. Once your creativity returns (with ideas on cool projects, the strong desire to read one the countless unread books on your shelf, people you really want to meet again, ...) you will feel more energetic than ever and can re-approach your life (and work) with a whole new mind.

4.3 Mentoring & Teaching

- Challenge the student by constantly asking questions. Let the student prove they got it. Set a high pace.

5 Programming

5.1 Common Principles

- Think hard before you start.
- Focus. Eliminate distractions and obstacles. If you cannot stay within *the flow*, you are doing something wrong.
- *Make it*, *make it run* (so that the tests pass), *make it better* (review and refactor until you can honestly say that it is good enough)
- If in doubt, leave it out (abstractions, features, ...), because *you ain't gonna need it (YAGNI)!* You can always come back later. Don't waste your time now.

5.2 Object-oriented Programming

- Using an OO-language does not imply you *'got it'* and are doing it right.
- If nothing else, do atleast try to follow the *tell, don't ask* principle.

5.3 Testing

- Look for *inflection points* as the sweet spot between testing effort and covered functionality.

5.4 Debugging

- Again, think before you start. Relax, breath and fetch your favorite beverage. Find the bug in your head. Only then start the debugger, if at all!
- Debugging is not a reason to stay overtime. Document your thought process so that you know where to pick up the next day.
- If stuck, describe your problem to an (imaginary) colleague.
- Dig until you have found the root cause. If you are inclined to just add a `null` check, you haven't found it yet.