PERFORMANCE-ORIENTED COMPUTING

Optimization (3) – Algorithms

GOALS

- Learn general **strategies** for algorithmic optimization
  - Optimizing individual algorithms isn't all that meaningful – the challenge is running into a **problem in some custom algorithm in software** you need to optimize
  - Try to provide generally applicable strategies to deal with this issue
- Discuss the potential of **memoization** for many algorithms
  - As an example of the general tradeoff between space and time

INTRODUCTION & OVERVIEW

OVERVIEW

- As we discussed before, algorithmic optimization is the **most important** category of optimization
  - Largest potential of performance improvement (without adapting requirements)
  - Broad spectrum of sub-categories

Most optimizations which touch data structures that we discussed before ***are*** *algorithmic optimizations!*

- Contains all optimizations which solve a problem in a **fundamentally different** way

CHALLENGES

- In practice, algorithmic optimization is **very challenging**
  - **Widely-used** and easily identified algorithms are **already optimized**
    - → May need to tune and select for specific target hardware
- *What if the algorithm* ***isn't*** *widely used or easily identified?*
  - Need to learn to recognize algorithmic similarities
  - Sometimes, *might* actually need to come up with a new algorithm
    - But more likely, a heuristic **based on** an existing algorithm and tuned for your use case

STRATEGIES FOR ALGORITHMIC OPTIMIZATION

1. Select **appropriate data structures** for your operations
   → We already discussed this selection process in depth in the previous chapter
2. Apply existing **best practices**
3. **Leverage** program-specific information and **context**

Some hints for these provided in this lecture.

ALGORITHMIC OPTIMIZATION STRATEGIES

And Some Examples

APPLYING BEST PRACTICES

This is often the hardest part!

- Essentially three steps:
  1. **Recognize** program task/process as *instance of an existing problem*
  2. **Study literature** to find best approach
  3. **Implement** – or better, find a suitable library or existing implementation

Can enable you to **benefit from** years or even decades of algorithm **research**, **optimization**, and in the case of libraries even implementation experience and **polish**.

HOW TO RECOGNIZE INSTANCES OF WELL-STUDIED ALGORITHMS

- Familiarize yourself with important algorithms, especially in **optimization** and **graph theory**
  - E.g. knapsack problem, maximum flow, …
- **Analyze the problem domain** and the nature of the data involved
  - Look for patterns, similarities, or **common structures** with known algorithms
  - Compare problem **characteristics**: objectives, constraints, input/output, …

This is a skill that improves with **experience** and exposure to various problem domains. Practice and continuous learning will enhance your ability to identify and apply appropriate techniques.

EXAMPLE:
REGISTER ALLOCATION IN COMPILERS

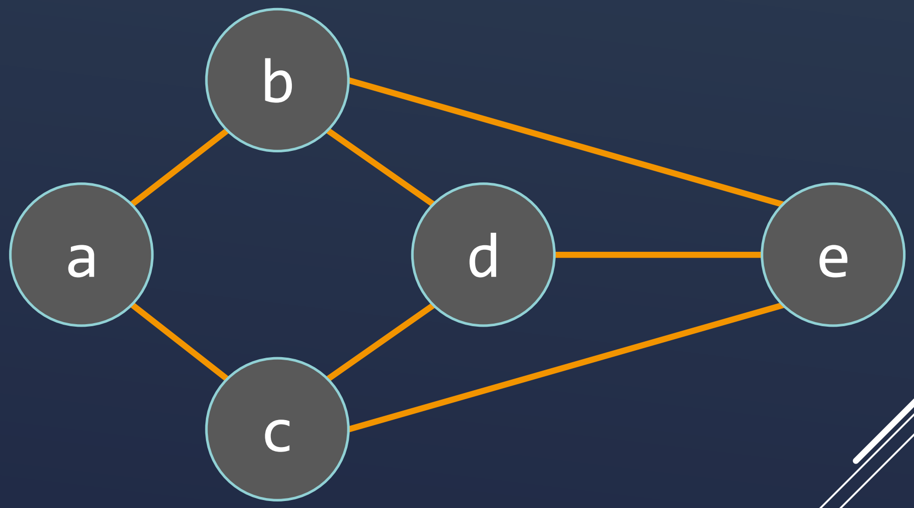
- Essential part of the **code generation** phase of compilers
- ***Goal***: use the limited set of available ISA registers as efficiently as possible to store the (potentially much larger) set of intermediate values

➔ Can be mapped to the **graph coloring** problem (specifically *vertex coloring*)

  - Limited set of resources (colors) need to be assigned to graph vertices
  - No two directly connected vertices may share the same color

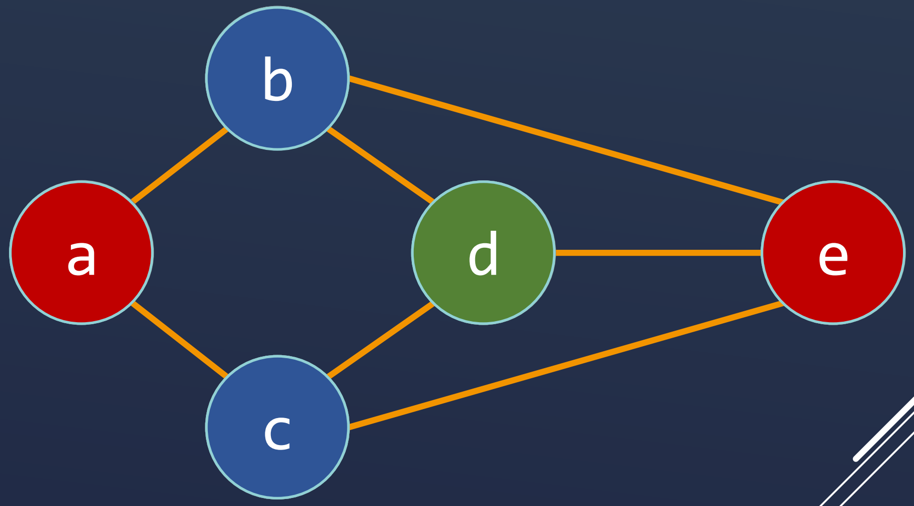
EXAMPLE:
REGISTER ALLOCATION IN COMPILERS

- ***Goal***: use the limited set of available ISA registers as efficiently as possible to store the (potentially much larger) set of intermediate values
- Make use of an *interference graph*
  - **Vertices**: temporary values to be stored
  - **Edges**: connect values required concurrently

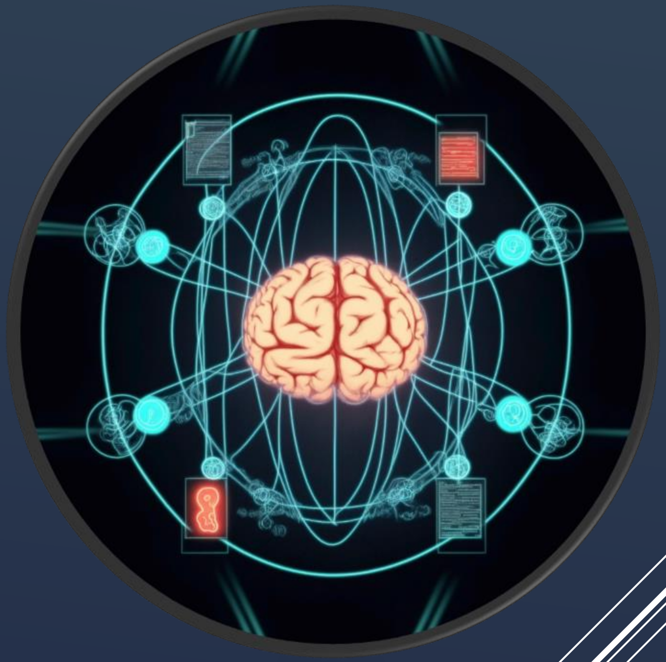


EXAMPLE:
REGISTER ALLOCATION IN COMPILERS

- ***Goal***: use the limited set of available ISA registers as efficiently as possible to store the (potentially much larger) set of intermediate values
- Make use of an *interference graph*
  - **Vertices**: temporary values to be stored
  - **Edges**: connect values required concurrently
- If graph can be colored with N colors
  → values can be kept in N registers!
- If fewer registers available
  → need *spilling* decision
  ➔ domain-specific heuristics



EXAMPLE:
REGISTER ALLOCATION IN COMPILERS



- *How would you think of this approach?*
- At least three important components:
  1. **Know about the graph coloring problem**
     → *"Familiarize yourself with important algorithms"*
  2. **Understand the core constraints at an abstract level**
     (i.e. value lifetimes and their interaction with limited register slots)
     → *"Analyze the problem domain"*
  3. **Formalize the problem in a structured way**
     (i.e. in this case as a graph)

➔ **General strategy** which applies to many problems and domains

LEVERAGING PROGRAM-SPECIFIC CONTEXT

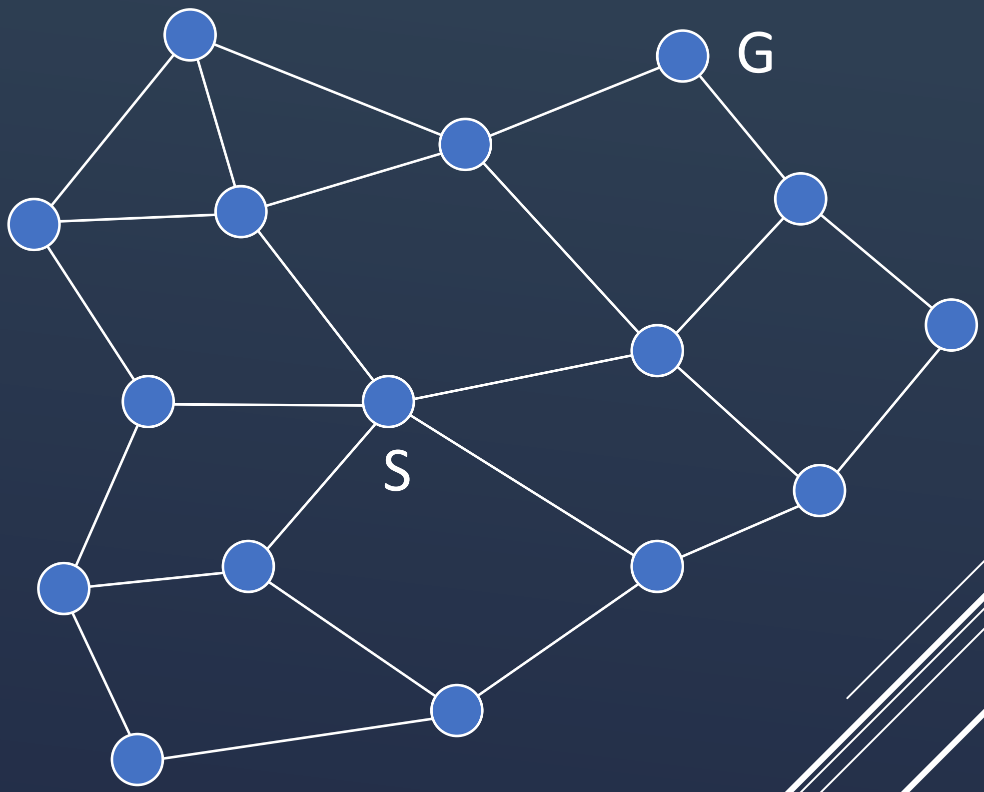
- The optimal **generic** algorithm may not be ideal in a **specific** situation / context in a given optimization target program

- Knowledge of the relevant **context** can be leveraged
  - E.g. range of input data, structure of objects, semantic constraints on results, …

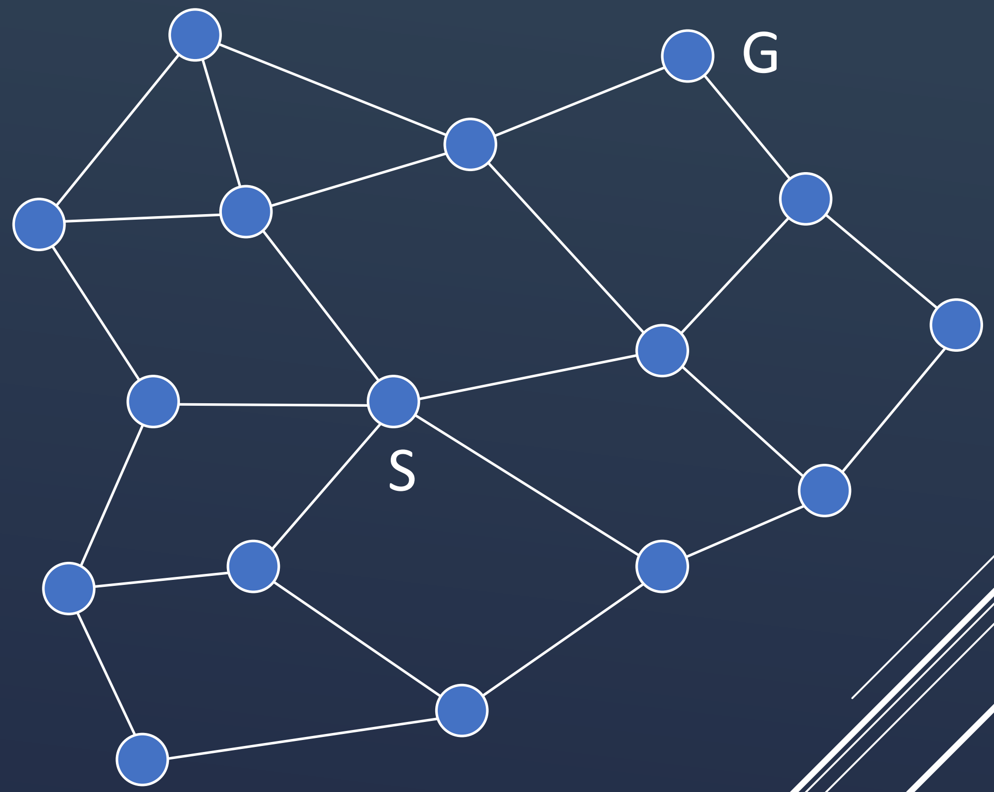
➔ In practice, this information is often explicitly or implicitly used in application- or **domain-specific heuristics**

EXAMPLE: A* PATHFINDING

- Dijkstra's Algorithm is **optimal** for the case of **generic** graphs
- However, we can achieve far better average-case performance in common **special cases** by **leveraging additional information**!
- **A*** uses the additional concept of a minimum possible distance from a given node to steer the search for the shortest path



EXAMPLE: A* PATHFINDING



- For this example:
  - Assume that graph is embedded in a conventional 2D space exactly matching the layout in the illustration

  → Use **Euclidian distance** as minimum!

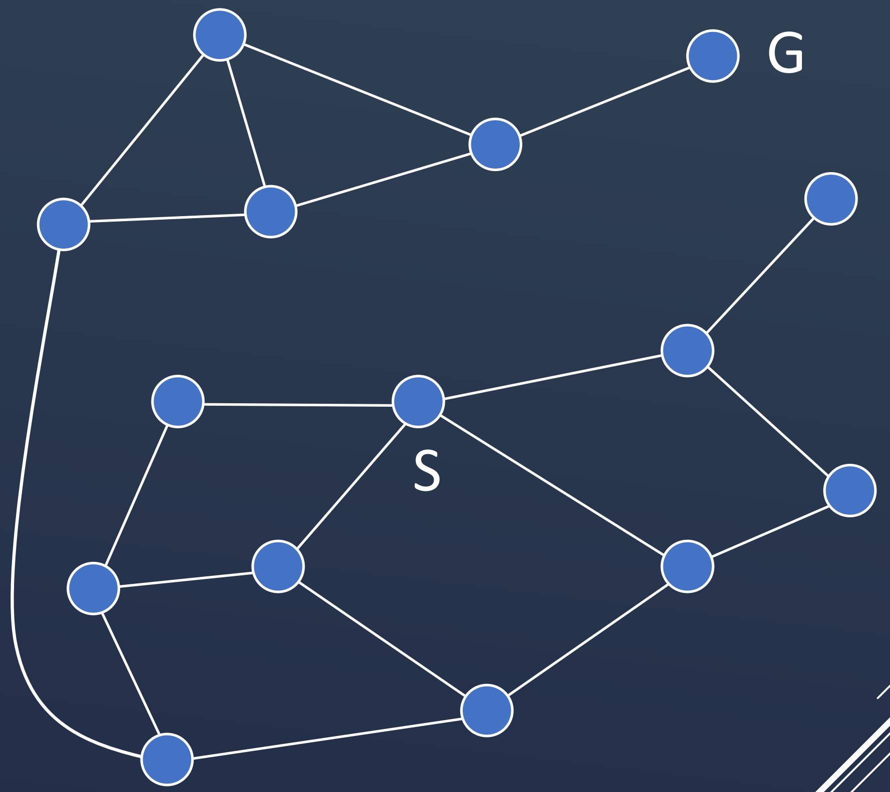
- ***Important***: A* is not a “better algorithm” in a general sense

  ➔ **Context** is essential!

EXAMPLE: A* PATHFINDING

- Heuristics are only as good as their assumptions
  - There are usually degenerate cases which fall back to worst-case complexity of the underlying algorithm
  - On the right you have an example of that for A*

- **Again**: understanding the exact **context** and statistical distribution of various scenarios in **your application** is essential



A NOTE REGARDING THE EXAMPLES

- You might have noticed that both of the examples here are based on algorithms in graph theory
- This is **not** because everything you'll encounter will reduce to some algorithm from graph theory with an efficient solution
  - It's primarily for educational purposes, because these graph algorithms are relatively easy to effectively **visualize** and explain

That said, a **lot** of problems ***do*** map really nicely to graph algorithms, and it is absolutely worth it to study the most important graph algorithms!

SUMMARY ALGORITHMIC OPTIMIZATION

- **Mapping** to well-studied problems
  - Perhaps the most **challenging** part, especially when it comes to less widely used algorithms
  - Requires knowledge of a **broad set** of common algorithms and approaches
  - Improves incrementally with **experience**
- Leveraging p**rogram-specific context**
  - Generally **requires** knowledge of the domain
  - Can make sense to discuss this with domain specialists

➔ In both cases, a **holistic view** of the program and its context is necessary!

MEMOIZATION

And the tradeoff between space and time

OVERVIEW

- **Memoization** is an optimization technique which can be extremely effective
- Basic idea:

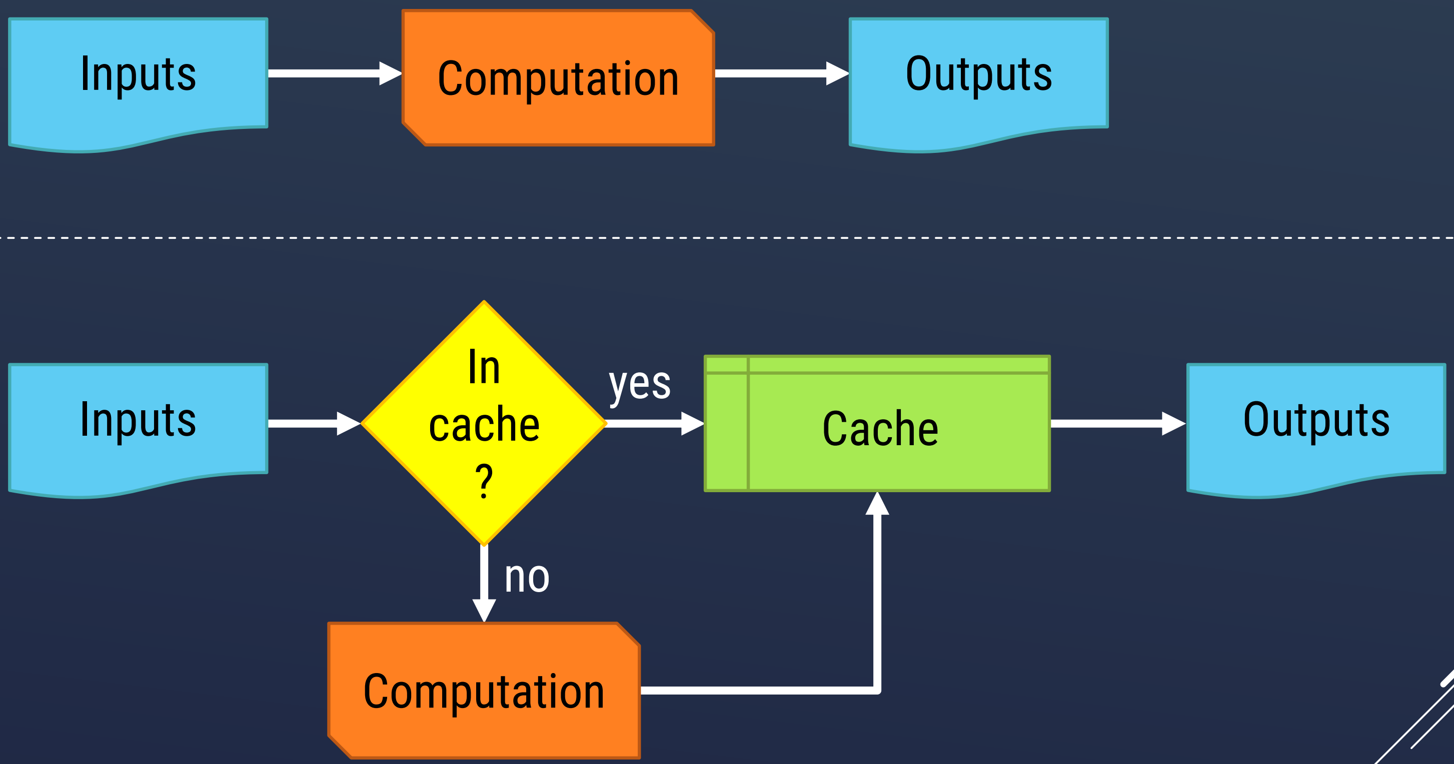
Store the results of expensive function calls and **reuse** them when the **same inputs** are encountered again.

- Avoids **redundant** computations **at runtime**
(rather than e.g. redundant expression elimination in the compiler)
  - Unlike redundancy elimination in the compiler, memoization can be applied with dynamic inputs and much more complex control flow

OVERVIEW

No Memoization

With Memoization



KEY CONSIDERATIONS

- **Most important**: memoization requires ***pure functions***
  - I.e. no side effects, no access to any inputs other than arguments

  → Same inputs must always produce the same outputs

- Especially for larger outputs, **cache invalidation / cleaning** strategies need to be employed
  - This can be very complex and increase the implementation effort while reducing effectiveness

CACHING STRATEGIES

- Primarily requires a **fast lookup** structure / operation
- Generally realized using an **associative array/map** backed by a **hash table**
  - Special cases might use a more algorithm-specific data structure, e.g. a simple dense array or specialized tree-based map
  - Input parameters serve as keys, output stored as values

PARALLELIZATION

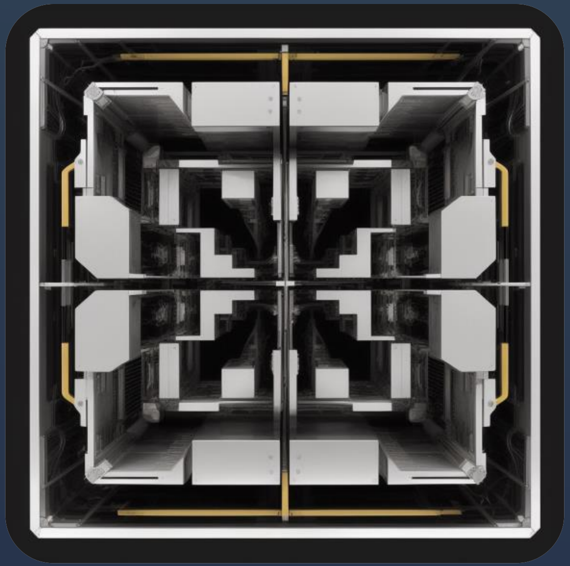
- **Note**: pure function calls can generally be executed freely and trivially in parallel **without** synchronization

➔ Caching can break this property!

- *Possibilities*:
  - **Per-thread Cache**
    - Requires even more memory, but retains synchronization-free property
  - **Read/write locked shared data structure**
    - Some overhead, but threads can share progress / results

Best choice depends on speed/**frequency** of individual computations, **size** of the input/output cache, and the expected **degree of parallelism**.

IMPLEMENTATION OPTIONS



Major options:

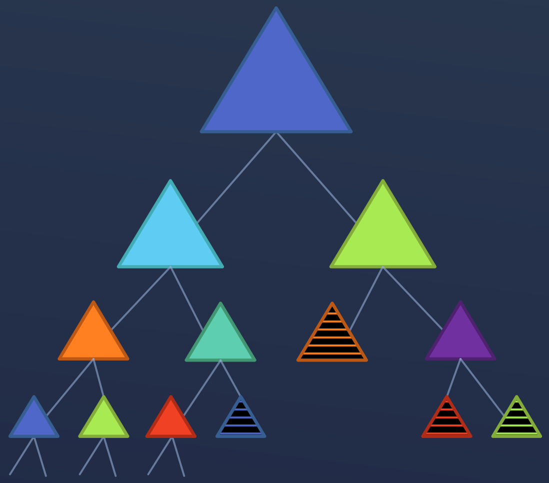
1. **Manually add caching** logic within the function
   - *Advantages*: potentially highest performance
   - *Disadvantages*: mixes actual algorithm logic with memoization implementation; harder to maintain and tune individually
2. **Wrap the function** in an interface which adds caching
   - *Advantage*: Clean separation; easy to read, maintain and tune
   - *Disadvantage*: might introduce mutual dependence in recursive case

In **both cases** a memoization **library or framework** can be used. Some languages might support simple function decorations to enable memoization.

SPECIAL CASE:
RECURSIVE DIVIDE-AND-CONQUER

- In **recursive divide-and-conquer** algorithms, memoization can be an **essential** part of the overall algorithm!

No memoization



With memoization

SPECIAL CASE: RECURSIVE DIVIDE-AND-CONQUER

- In **recursive divide-and-conquer** algorithms, memoization can be an **essential** part of the overall algorithm!
- Depending on the structure of data and degree of reuse, can **substantially reduce time complexity**
  - E.g. from exponential to quadratic or even linear in specific cases!
- This is one reason why **functional languages** sometimes have more first-class memoization support
- Particularly important application: **Dynamic Programming**
  - In this context, sometimes also called **tabulation**, mostly depending on how results are stored

MEMOIZATION TRADE-OFFS

- *Already discussed*: need for pure functions, **no side effects**, potential impact on ease and **effectiveness of parallelization**

Other performance / suitability considerations:

- **Size/variety** of the set of expected **inputs**
  - Larger input set
    → lower cache hit rate and more overhead for cache structure
- **Space vs. Time**
  - Larger keys and/or values
    → needs longer computations in order for caching to pay off
    → obviously requires more memory, and potentially more memory management

CONCLUSION

SUMMARY

- **Algorithmic Optimization**
  - Huge potential for performance improvement
  - Often challenging to apply
- **Strategies**
  - Applying best practices
  - Leveraging context and domain knowledge

→ Study **common algorithms** and gain a **holistic overview** of the program you are working on and its domain!

- **Memoization**
  - Idea & key considerations
    - Pure functions
  - Caching strategies
  - Parallelization
  - Implementation options
  - Trade-offs
- **Special Case: Recursion**
  - Dynamic Programming

QUESTIONS ?