Introduction to Algorithms and Data Structures: Bachmann–Landau notation

Giacomo Bergami
Newcastle University
http://jackbergus.github.io
ngb113@newcastle.ac.uk

Abstract

This tutorial will introduce you to the Bachmann–Landau notation, which is commonly used for analysing algorithmic computational complexity. You will see two different types of algorithms: ones that could be expressed via homogeneous linear recurrence relations with constant coefficients, and the other ones providing a balanced partitioning of the data data. For the first type of algorithms, we're going to discuss several different possible algorithms for obtaining the Fibonacci Sequence and, for the last one, we're going to discuss a specific type of metric tree, the *Vantage Point Trees*.

1 Introduction

1.1 Expressing Algorithms as Recurrence Relations

In this section, we will see how to represent a procedural problem over some data n into a recurrence relation, $F(n)$. As we will see later on, this will help us in defining a time recurrence relation $T(n)$ that will estimate how much time will it be required to compute $R(n)$.

As an use case example, let's talk about **Fibonacci Numbers**. Outside India, where such sequence was expressed as early as Pingala (c. 450 BC–200 BC), Fibonacci first uses the eponymous series to quantify the growth of rabbit colonies in his book "*Liber Abaci*" (1202). Considering the growth of a hypothetical, idealised (biologically unrealistic) rabbit population, Fibonacci assumes that:

1. a pair of rabbits generates two bunnies each year,
2. rabbits only start breeding in the second year after their birth, and
3. rabbits are immortal.

Given these considerations, we want to outline a function $F(n)$ that will predict the number of pairs existing at year n. In particular, we could generate the following algorithm to predict the bunny population in the following year:

Algorithm 1 Determining the next year population from the current year's distribution.

```
1: function GetNextYear(⟨mature, 1yo, newborn⟩)
2:     return ⟨mature+1yo, newborn, mature+1yo⟩
3: end function
```

Now, we want to define a function that, given a given year number n, is able to "predict" how many bunnies from all the ages are there. Before providing the informal proof of this, we might get first an intuition. Assuming that at the very beginning there is just one pair of bunnies ($n = 1$), we might get the folloing table:


Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

year	mature	1yo	newborn	**total**
1	0	0	1	1
2	0	1	0	1
3	**1**	0	***1***	2
4	1	*1*	1	3
5	*2*	1	*2*	5
6	3	2	3	8
7	5	3	5	11

By combining the previous pseudo-code and the current table, we might observe that, for each year $n \geq 3$:

- if there are b one-year pairs of bunnies, then in year $n-1$ there were b newborn pairs;
- the number of newborns is the same as the number of the mature bunnies: therefore, in year $n-1$ there were also b mature pairs of bunnies;
- if the number of mature bunnies is a, then it means that in year $n-1$ there were $a-b$ 1 year old bunnies that became mature.

As a consequence, we might define the following algorithm for returning the population distribution in the previous year:

Algorithm 2 Determining the previous year population from the current year's distribution.

```
1: function GetPrevYear(⟨mature, 1yo, newborn⟩)
2:     return ⟨1yo, mature-1yo, 1yo⟩
3: end function
```

We can now informally prove[1] the following lemma providing the recurrence relation for the Fibonacci Numbers:

▶ **Lemma 1.** *The Fibonacci numbers can be expressed with the following recurrence relation:*

$$\forall n \in \mathbb{N}\backslash\{0\}.\ F(n) = \begin{cases} 1 & n \leq 2 \\ F(n-1) + F(n-2) & \text{oth.} \end{cases}$$

Proof. Informal proof for a year $n+1$: from the two former algorithms, we know that GetNextYear$(\langle a, b, c\rangle) = \langle a+b, c, a+b\rangle$ and that GetPrevYear$(\langle a, b, c\rangle) = \langle b, a-b, b\rangle$. We can observe that GetPrevYear$(\langle a, b, c\rangle) + \langle a, b, c\rangle = \langle a+b, a, b+c\rangle$, thus given a total number of bunnies that is the same of the one provided by GetNextYear$(\langle a, b, c\rangle)$. ◀

We are now ready to express such function as a recursive function. At this point, we're able to define a simple recursive function in C++, but what about lambdas?

```
unsigned long fibonacci(unsigned long n) {
       return (n<=2) ? 1 : fibonacci(n-1)+fibonacci(n-2);
}
```

[1] We will introduce the proof by induction later on on these handouts. For the moment, it is important to understand why the following passages are correct, and why they provide the desired result.

1.2 C++ Recursive Lambdas

C++ lambdas have their origins in theoretical computer science, and more precisely in *lambda calculus*. Mathematician Alonzo Church in the 1930s introduced the lambda calculus as part of an investigation into the foundations of mathematics. The following terms define lambda calculus:

$$T ::= \lambda x.T \mid T\,T' \mid x \qquad x \in \text{Var}$$

By reading this grammar from left to right, we can see that we can express unnamed functions having a parameter x and having a body T ($\lambda x.T$), we can express applications (i.e., pass parameters to a function), and then we can represent variables (x). We can prove that this language is Turing Complete.

As you might notice, this simple language (as well as the lambda functions in C++) do not allow recursive functions: in fact, each function is unnamed, and therefore it has no "name" to call itself by. Nevertheless, in such simple language, it is possible to define a **fix-point operator** $\mathsf{fix}\ f = f\ (\mathsf{fix}\ f)$ also called Y, which repeated application allows us to do an arbitrary number of recursive calls:

$$\mathsf{fix}\ f = f\ (f\ (\ldots f\ (\mathsf{fix}\ f)\ldots))$$

If we want to express fix in C++, we shall define it as a struct accepting the function f as an argument. For the moment, we can start declaring such operator as follows:

```
template <class F> struct y_combinator {
    F f;
    // <function call operator>
}
```

By doing so, we define an implicit `y_combinator` constructor taking a (e.g., lambda function) of type F, and setting it to the field $f : F$[2].

It is easy to see how $\mathsf{fix}\ f$ may be applied to one variable. Applying it to two or more variables requires adding them to the equation:

$$\mathsf{fix}\ f\ x = f\ (\mathsf{fix}\ f)\ x$$

This means that any recursive lambda should accept a copy of itself as one of the arguments ($\mathsf{fix}\ f$), as well as the parameters x through which the function itself is called. So, we need to "call" our combinator instance with the same parameters as the function f as well as an instance of **`this`** mimicking the recursive function call. Therefore, the struct should also have a *function call operator*[3] taking all the arguments $\vec{x}\ldots$ required by f, and returning $f(\textbf{this}, \vec{x}\ldots)$:

```
// <function call operator> :=
// a forwarding operator(), returning the same type as the one by f
template <class... Args> decltype(auto) operator()(Args&&... args) const {
// we pass ourselves to f, then the arguments.
// the lambda should take the first argument as ‘auto&& recurse’.
return f(*this, std::forward<Args>(args)...);
}
```

In this code we might observe the usage of **`decltype(auto)`**: despite the fact that C++ allows programmers to use functional programming with static type check, the language

[2] It reads "f of type F".
[3] `https://en.cppreference.com/w/cpp/language/operator_other`

has not been designed to do inference typing with unknown parameters, such as in Haskell or OCaml. Therefore, when the type is defined via templates (generic code), you want to perfectly forward a return type without knowing whether you are dealing with a reference or a value. So, a decltype(auto) allows to correctly instruct the compiler to take the returning type of the Fun, that is in here implicitly defined as a (lambda) function.

Last, we can define a template function generating the recursive lambda function as follows:

```
/**
* Returning the lambda function after the decay, providing the lambda typecast
* @tparam F   The lambda type of the associated function: this lambda function must
    ↪ accept itself as a first auto parameter
* @param f    the actual lambda function
* @return     The closure via the y_combinator typecasted as a lambda function
*/
template <class F> y_combinator<std::decay_t<F>> make_y_combinator(F&& f){
return {std::forward<F>(f)}; // Calling the implicit constructor of the
// y_combinator. Forward is here used to ease
// typecasting issues and for ensuring to pass the correct type, no
// matter what value is passed to the parameter (lvalue or rvalue).
}
```

Now, we can declare our recursive lambda function as follows:

```
// <recursive Fibonacci> :=
auto fibonacci_recursive = make_y_combinator(
       [](auto&& fibonacci_recursive, unsigned long x) -> unsigned long {
  if(x <= 2)
    return x;
  else
    return fibonacci_recursive(x-1) + fibonacci_recursive(x-2);
});
```

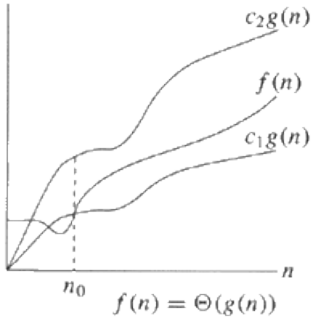
2 Bachmann-Landau notations

If you now try to run both versions of the Fibonacci function so far, you will notice that, for a huge long, the function will take quite a while to compute. At a first glance, you might suppose that it has something to do with recursion, and that's why it is going to take a while to compute. On the other hand, as we will able to see at the end of this lecture, it will all depend on how we're partitioning the data within each recursive call.

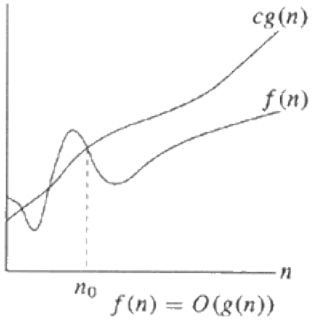
Beside the benchmarking evaluating the efficiency of a given implementation, we might also want to determine the efficiency of the solution regardless of the specific implementation of choice. To do so, we might need to analyse our algorithms more formally.

Asymptotic notations are used to describe a function's limiting behaviour when the argument tends to a specific value (often infinity), usually in terms of simpler functions. The most common notation of choice is the one after Edmund Landau and Paul Bachmann, that is hence called **Bachmann–Landau notation**. In this module we're only going to see two most common operators, those are Θ and O (Figure 1 on the facing page).

Before providing the formal definition of such operators, let us observe how such operators work in practice. With respect to the big-O, if the algorithm has computational complexity $O(T_1(n) + T_2(n))$ where $\forall n.T_1(n) \geq T_2(n)$, then $T_1(n) + T_2(n) \in O(T_1(n))$; this means that additive (as well as multiplicative) constants are negligible, and that if an algorithm takes $n + n^2$ time to compute, then we can represent such computational complexity as $O(n^2)$. In particular, constant values might represent "immediate" operations, such as the lookup of a key in a hash-map, reading a primary memory location at a given offset, performing



(a) Function f is "strictly bounded" (inner and upper limit) by g times a constant multiplicative factor.

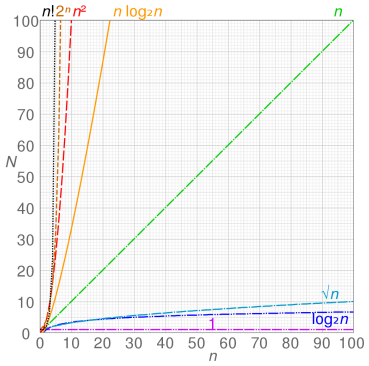


(b) Function f is "upper bounded" by g times a constant multiplicative factor.

Figure 1 Intuitive explaination on how the O and Θ operators works.

Big-O	Name
$O(1)$	constant.
$O(\log\log n)$	double logarithmic.
$O(\log n)$	logarithmic.
$O(\log^c n)$	polylogarithmic.
$O(n^c)$	for $0 < c < 1$, fractional power.
$O(n)$	linear.
$O(n\log n) = O(\log n!)$	loglinear.
$O(n^2)$	quadratic.
$O(n^c)$	for $c > 2$, polynomial.
$O(e^x)$	exponential.
$O(x!)$	factorial

(a) Names of the most common order functions.



(b) Plotting order functions.

Figure 2 Intuitive explaination on how the O and Θ operators works.

an elementary mathematical operations, returning a value. On the other hand, Θ reflects a more strict bound over the function, representing a strict bound where just additive and multiplicative constants are discarded. Figure 2 provides a list of the most common representations of the order functions, and a graphical representation of such functions. We can now provide a more formal definition of such operators, which we are going to need to prove the computational complexity of the previous Fibonacci "algorithm".

▶ **Definition 2** (Θ). *If an algorithm F is of $\Theta(T(n))$, it means that the running time of the algorithm as n (input size) gets larger is proportional to $T(n)$:*

$$0 < \lim_{n\to\infty} \frac{F(n)}{T(n)} < \infty \quad \exists k_1, k_2 > 0, n_0.\ \forall n > n_0.\ k_1 T(n) \leq F(n) \leq k_2 T(n)$$

▶ **Definition 3** (O (big-O)). *If an algorithm F is of $O(T(n))$, it means that the running time of the algorithm as n gets larger is at most proportional to $T(n)$:*

$$\lim_{n\to\infty} \frac{F(n)}{T(n)} < \infty \qquad \exists k > 0, n_0.\ \forall n > n_0.\ |F(n)| \leq kT(n)$$

While Bachmann-Landau notations provide a compact notation for classifying algorithms with respect to their computational complexity, this notation is not able to differentiate

solutions using data structures differing by just a constant factor. For example, read and write constants might be differently weights due to locks, as well as performing a linear scan of a vector is more efficient than doing large memory jumps for scanning double-linked lists. In all these scenarios where constants might be crucial, we might need to define a different computational model as well as ignoring asymptotic notations.

2.1 Fibonacci Sequence

Let us analyse the definition of our recursive Fibonacci "algorithm": after observing that the function F matches with the computational cost T to provide the computational complexity value, we can observe that T represents a **homogeneous linear recurrence relation with constant coefficients**. This means that the general form of T is $\sum_{1\leq i\leq h} c_i T(n-i)$ in the non-recursive (or *base*) cases for some constants c_i. In this case, each constant c_i represents how many times the i-the recursive function call is called. We might ask ourselves is there is a way to automate the inference process for the overall computational complexity.

▶ **Lemma 4** (Homogeneous linear recurrence relation with constant coefficients)**.** *Given some $a_1, \ldots, a_h \in \mathbb{N}$, with $h \in \mathbb{N}\backslash\{0\}$, $c > 0, \beta \geq 0 \in \mathbb{R}$, and given $T(n)$ the following recurrence relation:*

$$T(n) = \begin{cases} \text{const.} & n \leq m \leq h \\ \sum_{1\leq i\leq h} a_i T(n-i) + cn^\beta & n > m \end{cases}$$

if $a = \sum_{1\leq i\leq h} a_i$, then [3]:

- *if $a = 1$, then $T(n) \in O(n^{\beta+1})$*
- *if $a \geq 2$, then $T(n) \in O(a^n n^\beta)$*

We might observe that in our algorithms $a = 2$ and that $\beta = 2$, and therefore we can easily derive that the algorithm's computational complexity is $O(2^n)$. Appendix A on page 21 will provide a formal proof on how to demonstrate such result without using the theorem.

3 Improving algorithmic efficiency

In this section we might investigate two different approaches for optimizing an algorithm represented as a recurrence relation: we might either try to solve mathematically the recurrence relation so to achieve near-constant computational complexity or, if we cannot do that, we can store partially-computed results for future computations. In order to show these two different approaches, we will continue using the Fibonacci recurrence formula as an use case.

3.1 Binet's Formula

Like every homogeneous linear recurrence relation with constant coefficients, the Fibonacci numbers can be represented by a a mathematical expression that can be evaluated in a finite number of operations (*closed-form solution*). The closed-form solution associated to the Fibonacci is the following:

$$F(n) = \frac{\sqrt{5}}{5}\left(\left(\frac{\sqrt{5}+1}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

If we assume that the power is computed in constant time then this formula can definitively be computed in constant time, thus providing an infinite[4] speedup. In this case the associated lambda function is very simple, and it can be written as follows:

```
// <Binet's formula> :=
auto binet = [](unsigned long n)->unsigned long {
    // computing the golden ration number only once. Using double precision
    // to reduce numerical errors.
    static const double phi = (1 + std::sqrt(5))*0.5;
    // computing the associated mathematical expression
    double fib = (std::pow(phi,n) - std::pow(1-phi,n))/sqrt(5);
    // rounding the result.
    return round(fib);
}
```

In the remaining part of the section, we're going to discuss how to obtain such formula from the initially given recurrence relation. We can prove[5] that a homogeneous linear recurrence relation of arbitrary order with $\beta = 0$ and $c = 1$ have a characteristic polynomial $p(\lambda) = \lambda^h - \sum_{1 \leq i \leq h} c_i \lambda^{h-i}$ that can be obtained as $\det(\lambda I_h - \mathbf{A})$ from the linear recurrence matrix $\mathbf{A}$ defined as follows:

$$A_{ij} = \begin{cases} c_j & i = 0 \\ 1 & i + 1 = j \\ 0 & \text{oth.} \end{cases}$$

The (distinct) roots $\{\lambda_i\}_{1 \leq i \leq h}$ of such polynomial will be hence the *eigenvectors* of $\mathbf{A}$ and the elements of the diagonal matrix $\mathbf{D}$ defined as $D_{ii} = \lambda_i$. Then, after finding the eigenvectors $\mathbf{v}_i$ associated to each eigenvalue λ_i, then we can find a matrix $P = [\mathbf{v}_1 \ \dots \ \mathbf{v}_h]$ such that $\mathbf{A} = P\mathbf{D}P^{-1}$. After noticing that $\mathbf{A}^n = P\mathbf{D}^n P^{-1}$, we can just expand the matrix multiplication for then obtaining the required coefficients.

Given the vector $u_{n-1} = [F(n-i)]_{1 \leq i \leq h}$, we can now express u_n as $u_n = \mathbf{A}u_{n-1}$. E.g., for the Fibonacci sequence, we will have $A = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$ and $u_n = \begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix}$. In fact, we can see that such conditions allow us to express the following linear system:

$$\begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F(n-1) \\ F(n-2) \end{bmatrix} \Rightarrow \begin{cases} F(n) = F(n-1) + F(n-2) \\ F(n-1) = F(n-1) \end{cases}$$

We can obtain each eigenvector $\mathbf{v}_i$ by solving the linear system $(A - \lambda_i I)\mathbf{v}_i = 0$. After that, we will obtain a matrix $P = \begin{bmatrix} 1 & 1 \\ \frac{\sqrt{5}-1}{2} & -\frac{1+\sqrt{5}}{2} \end{bmatrix}$: we will leave to the reader the computation of P as an exercise. Given that we are only interested in the first part of the linear system providing me the solution of the recurrence relation, we will obtain:

$$u_n = A^n u_0 = \begin{bmatrix} 1 & 1 \\ \frac{\sqrt{5}-1}{2} & -\frac{1+\sqrt{5}}{2} \end{bmatrix} \begin{bmatrix} \frac{\sqrt{5}+1}{2}^n & 0 \\ 0 & \frac{1-\sqrt{5}}{2}^n \end{bmatrix} \begin{bmatrix} 1 & 1 \\ \frac{\sqrt{5}-1}{2} & -\frac{1+\sqrt{5}}{2} \end{bmatrix}^{-1} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

After expanding all the mathematical steps, then we will obtain Binet's formula.

---

[4] The *speedup* is defined by the limit towards $+\infty$ of the ratio between the baseline algorithm and the improved solution, thus $\lim_{x \to +\infty} \frac{e^x}{x} = \infty$

[5] As this part is not required, we will leave the reader to deep dive into the formal proof that is available at `https://artofproblemsolving.com/wiki/index.php/Characteristic_polynomial`. See also `https://math.hawaii.edu/~rharron/teaching/MA142/RecurrenceRelations.pdf`.

3.2 Memoization

We've seen before that some recurrence relations might be always expressed via a mathematical expression, which evaluation can be computed in nearly constant time. Nevertheless, in some cases we might not just be able to compute a mathematical equation to be solved in constant time. When an algorithm $F(n)$ requires computing a same subset of sub-problems for different inputs n, then we might want to **cache** those intermediate results for later re-use. In particular, a cache is a hardware or software component that stores data so that future requests for that data can be served faster. For example, the operative system caches the data and program pages so that the ones that are more requested will be promptly provided to the CPU (e.g., LRU) [10, 12]. An explicit way to exploit caching is to use **memoization**, which is a specific form of caching that involves caching the return value of a function based on its parameters. The memoization principle can be expressed by the following pseudocode:

Algorithm 3 Generic memoization scheme.

```
1: global map = ();
2: function MEMOIZATION(F, x)
3:     if x not in map.keys() then
4:         map[x]=F(x)                          ▷ Cost: T(n) + O(1)
5:     end if
6:     return map[x]                            ▷ Cost: O(1)
7: end function
```

As the function shows, we need to compute each subproblem $F(n)$ only the first time it occurs while, on all the following requirements, we just want to pay $O(1)$. This can be achieved if we store the value in a hashmap, so that the lookup for n can be achieved in the required computational complexity. If we assume to define the memoization map as a field `memoization` in a class and we want to declare the lambda in its constructor, then we can express our recursive lambda function with memoization as follows:

```
// <memoized lambda> :=
auto memoized = make_y_combinator([this](auto&& rec, unsigned long x)->unsigned long {
  if (x<=1) return x; else {
    auto f = memoization.find(x);
    if (f != memoization.end()) return f->second; else {
      unsigned long value = rec(x-1) + rec(x-2);
      memoization.emplace(x, value);
      return value;
    }}})
```

In order to better outline such solution, we need to check what will it happen in the best case scenario, and what might happen in the worst case scenario. We might have the best case scenario when $x \leq 2$ and therefore we only "pay" $O(1)$, or we might get the best scenario if $F(n)$ was already computed in a previous step: in this case, the map is accessed via the key n and the associated value returned. As a consequence, the worst case scenario might happen only when $n > 2$, when n is sufficiently big and when the memoization map is empty. we can prove that in this case we're going to achieve a linear computational complexity. Again, we need to prove this statement by induction.

▶ **Lemma 5.** *In the previously outlined worst case scenario, if we assume that expressions are interpreted from left to right, then the cost function T_m of Fibonacci with memoization over n is in $O(n)$.*

Proof. We can prove the base case for $n = 3$ and, just to confirm this assumption, we can also check that the condition holds even for $n = 4$ and $n = 5$. Please consider that, for each n, we're always going to assume that the memoization map is empty. For $n = 3$ the condition is proved because we're going to perform two recursive call which, fortunately, are going to return constant values. We have:

$$T_m(3) = 2c_0 \leq 3c_0$$

For $n = 4$, then we will need to compute $T_m(3)$ without memoization, and then compute $T_m(2)$ which is never memoized because $F(2) = 1$. So:

$$T_m(4) = T_m(3) + T_m(2) \leq 4c_0$$

The next base case will explicitly show for the first time the usage of memoization: for $n = 5$, then we will have $T_m(5) = T_m(4) + T_m(3)$, where $F(4)$ is computed and then memoized, while the value for $F(3)$ is directly returned by the memoization map, because it was already computed as a part of the computation for $F(4)$. In this case we'll have that $T_m(5) = T_m(4) + T_m(3) \leq 4c_0 + T_m(3) = 5c_0$, and therefore even in this case the property is proved.

Let us now examine the induction case: for all $n' < n$, if we assume that $T(n') \leq c_0 n'$, then we need to prove that also $T_m(n) \leq c_0 n$. In order to prove that, we can use the inductive assumption and hence return the following result as expected:

$$T_m(n) = T_m(n-1) + T_m(n-2) \leq c_0(n-1) + T_m(n-2) = c_0 n$$

where $T_m(n-2) = c_0$ per memoization. ◀

The major problem with memoization is that, for many computations and many possible partial memoized computations, such solution might require too much primary memory. In these situations it's recommended to use columnar or key-value databases, such as RocksDB[6].

3.3 Benchmarking

We want now to show that these computational complexity analyses are correct by performing a benchmarking over such lambdas implementations. In order to do so, we might design a class `fibonacci_no_master_theorem` containing: *(i)* an enumeration associating each lambda to each possible algorithmic implementation:

```
// <enum over the possible lambdas> :=
enum algorithm {
RECURSIVE,
RECURSIVE_WITH_MEMOIZATION,
BINET_FORMULA,
};
```

(ii) a constructor associating the enum to each lambda via an unordered map, as well as the memoization map:

```
class fibonacci_no_master_theorem {
public:
// <enum over the possible lambdas>
```

[6] `https://github.com/facebook/rocksdb/`

```
fibonacci_no_master_theorem();
void fibonacci_no_master_theorem::clearMemoization(){memoization.clear();}
// <rest of the implementation>

private:
std::unordered_map<algorithm, std::function<unsigned long(unsigned long)>> map;
std::unordered_map<unsigned long, unsigned long> memoization;
}
```

The constructor can be implemented in the `cpp` file as follows:

```
fibonacci_no_master_theorem::fibonacci_no_master_theorem() {
    // <recursive Fibonacci>
    map.insert(std::make_pair(RECURSIVE, fibonacci_recursive));

    // <memoized lambda>
    map.insert(std::make_pair(RECURSIVE_WITH_MEMOIZATION, memoized));

    // <Binet's formula>
    map.insert(std::make_pair(BINET_FORMULA,binet));
}
```

(iii) a function that allows to compute the Fibonacci functions with the algorithm of choice:

```
// <rest of the implementation> :=
unsigned long get(algorithm choice, unsigned long x) {
  auto f = map.find(choice);
  if (f == map.end()) return 0; else
  return f->second(x);
}
```

At this point, we can define a `doBenchmark` function that is going to perform the benchmark in the same way for every algorithm. To get the actual process running time, thus excluding the interleaving time, we're going to count how many processor ticks occurred for each program. To do so, we're going to use the clock from the new C++ standard library.

```
#include <chrono>
typedef std::chrono::high_resolution_clock Clock;
```

Now, we can define a single benchmarking function that will take a baseline and compare it with all the other algorithms. The baseline algorithm's time will be 1, so that the ration between any different algorithm and the running time of the baseline will give the slowdown factor (i.e., the inverse of the speed-up). Such generic function needs to do the following steps:

- For each algorithm, run the function over `MAX_ITERATIONS` inputs and, for each input x, do `SAMPLES` runs. Record the running time for all the `SAMPLES` iterations and do the average of the resulting time.
- If I'm benchmarking first the baseline algorithm that we assume (just for the moment) that has constant computational time, then we'll do the `average` of all the possible clock times.
- If I'm benchmarking the non-baseline algorithms, then divide the averaged running time for each iteration by the `average` value from the baseline function.
- If the benchmarking algorithm considers memoization, then clear the map after each sample.
- For each algorithm's iteration, print the result into screen, so that we can plot it with a program of choice.

The resulting function is the following:

```
void doBenchmark(fibonacci_no_master_theorem &factory, double& average,
    ↪ fibonacci_no_master_theorem::algorithm x, bool clearMemoization, bool doBaseline
    ↪ ) {
      for (unsigned int i = 1; i < MAX_ITERATIONS; i++) {
        auto before = Clock::now();
        for (unsigned int j = 0; j<SAMPLES; j++) {
          factory.get(x, i);
          if (clearMemoization) factory.clearMemoization();
        }
        auto t = (Clock::now() - before)/(SAMPLES);
        if (doBaseline) {
          std::cout << t.count(); average += t.count();
        } else
          std::cout << round(t.count()/average);
        if (i == MAX_ITERATIONS-1)
          std::cout << std::endl;
        else
          std::cout << ",";
      }
      if (doBaseline)
        average /= MAX_ITERATIONS;
}
```

Now, we can call the function for all the possible algorithms in the following way:

```
int main() {
fibonacci_no_master_theorem factory;;
double average = 0;

// Binet's formula
doBenchmark(factory, average, fibonacci_no_master_theorem::BINET_FORMULA, false, true);

// Using recursion with memoization just for the first number, so to observe the worst
    ↪ case scenario
doBenchmark(factory, average, fibonacci_no_master_theorem::RECURSIVE_WITH_MEMOIZATION,
    ↪ true, false);

// Using recursion with memoization, starting from the first element, so to observe the
    ↪  best case scenario
doBenchmark(factory, average, fibonacci_no_master_theorem::RECURSIVE_WITH_MEMOIZATION,
    ↪ false, false);

// Recursive function with no memoization (purely recursive)
doBenchmark(factory, average, fibonacci_no_master_theorem::RECURSIVE, false, false);

return 0;
}
```

Figure 3 provides the log-plot of the algorithms' clock running time. We were forced to plot the logarithm of the time clock due to the exponential running time of the recursive function. In this log plot, the exponential function is rendered as a line[7], the linear memoization function where the map was cleared at each step is represented by a logarithmic function, and the non-cleared memoization function that is run from the minimum element to the maximum one still appears as a constant function. This benchmark then validates our theoretical proves and assumptions.

[7] Remember that $\log(e^x) = x$

4 Use Case: Vantage Point Tree

In this section, we're going to use **Vantage Point Trees** (VP-Trees) as a use case for introducing a new type of recurrence relations (§4.1), and for exploiting transparent caching while accessing and visiting binary trees (§4.2). Last, we're going to introduce the Vantage Point Tree as a specific case of a metric tree (§4.3).

4.1 Introduction: Recurrence relations with balanced partitioning

In the previous section we saw that, if the recurrence relation input is decreased at each recursive call by just a constant factor, we might come up with a polynomial ($a = 1$) or an exponential ($a \geq 2$) algorithm. At this point, we want to uniformly partition the data and dividing the problem into smaller sub-problems of the same size. This general technique is called **divide and conquer**, and consists in the following phases [6]:

- **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
- **Conquer** the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
- **Combine** the solutions to the subproblems into the solution for the original problem.

Divide and conquer algorithms share more or less the same following schema:

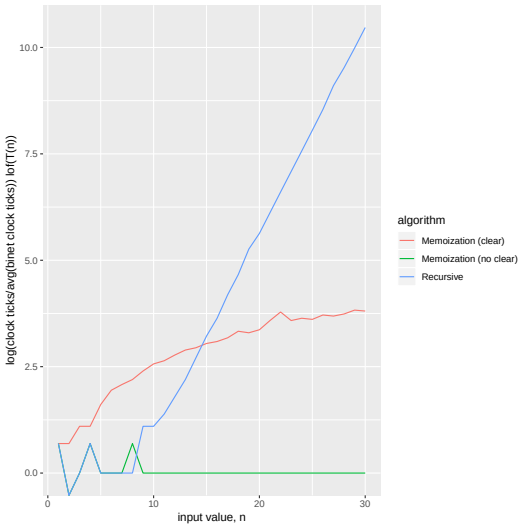


Figure 3 log-plot of the clock time of the algorithms' running time.

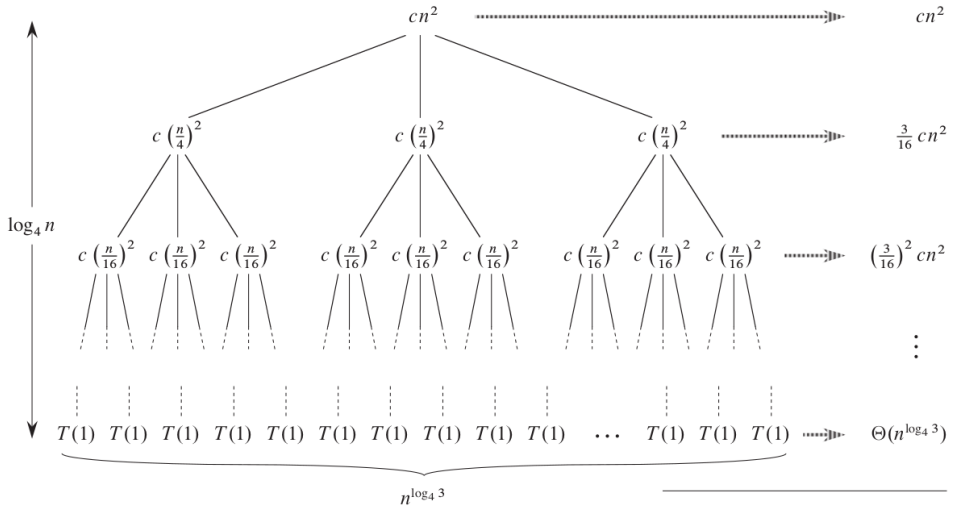


Figure 4 Recursion tree for $T(n) = 3T(\frac{n}{4}) + cn^2$: given that $c = 2 < \log_4 3$, then this example falls in the first case of the master theorem, thus $T(n) \in \Theta(n^{\log_4 3})$ [6].

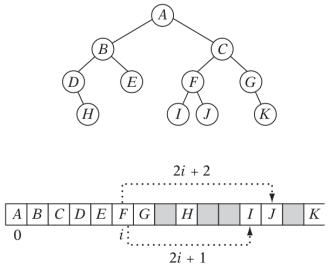
Algorithm 4 Generic schema of a Divide and Conquer algorithm

```
1: procedure DAC(x, σ, f)
2:     n := |x|
3:     if n < k, k constant then
4:         return σ(x)                                   ▷ Solve x directly
5:     else
6:         Create a subproblems in S, each of those has size n/b.   ▷ |S| = a
7:         return f({ DAC(x) | x ∈ S })   ▷ Combine each resolved subproblem via f(x)
8:     end if
9: end procedure
```

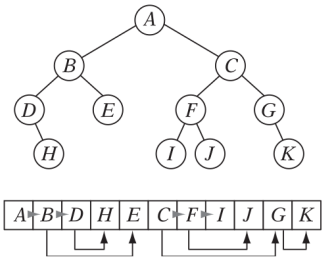
We can graphically visualize the merge of all the *stack calls* of our algorithm DAC as a **recursion tree**, where each node at the i-th level (counting the root as zero) represents the computational complexity of computing a solution of size $\frac{n}{b^i}$: in this case, a will correspond to the branching factor of the tree. Each leaf of the tree will be necessarily a constant size solution δx computed via $\sigma(\delta x)$. As evidenced by the theorems solving the recurrence relations, the sum of the cost of all those leaves will provide as a total a non-constant function.

As an example, let's consider a generic algorithm with cost $T(n) = 3T(\frac{n}{4}) + cn^2$ (Figure 4): The top node has cost cn^2, because the cost of splitting x into subproblems S is considered as negligible, while the whole computational cost is dominated by the cn^2 units of work required by σ to recombine the partial results. As per previous observations, at each immediately following recursive call, the data-size is split by 4, while the function is recursively called 3 times on (possibly different) subsets of the data. In order to determine the computational complexity of this algorithm, we're going to apply the result from the **master theorem** directly:

▶ **Theorem 6** (Master Theorem). *Let $a \geq 1, b > 1, d > 0 \in \mathbb{N}$ be constants, $f(n)$ a non-*



(a) Parent-child data are not particularly convenient for non-balanced trees. On the other hand, we might avoid to store child node pointers.



(b) Preorder trees: grey edges refer to `boolean` values determining whether the left subtree exists, while the pointers refer to the right subtree.

Figure 5 Different representations for primary-memory cache efficient trees[8].

negative function, and $T(n)$ *be defined by the following recurrence [6, 3]:*

$$T(n) = \begin{cases} d & n = 1 \\ a \cdot T(\frac{n}{b}) + f(n) & n > 1 \end{cases}$$

Then, $T(n)$ *has the following asymptotic bounds:*

1. *When* $f(n) = O(n^c)$ *with* $c < \log_b a$*, then* $T(n) = \Theta(n^{\log_b a})$.
2. *When* $\forall k \geq 0. f(n) = \Theta(n^{\log_b a} \log^k n)$*, then* $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. *If* $f(n)$ *is not dominated by* $n^{\log_b a + \epsilon}$ *asymptotically* (Ω) *with* $\exists c < 1. a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$*, then* $T(n) = \Theta(f(n))$.

Case 2 can be extended for any possible logarithmic power [13].

4.2 Optimizations: Principle of Locality

As previously discussed, in some cases the additive and multiplicative constants are critical, especially for "big data". In these cases, we can directly express the algorithm's computational complexity with no asymptotic notation. Given that memoization strategies might be critical in such situations, thus including more tortuous and hardly-readable source code, we might prefer to exploit a *transparent caching* system [2]. To avoid possible cache misses from accessing the information either accessing sparse memory information or fragmented information within a file [10, 12], we then exploit the tendency of a processor to access the same set of memory locations repetitively over a short time (**principle of locality**). Among these three approaches, we can distinguish three ways to achieve the principle of locality:

1. **Temporal locality**: reusing the same data within a short period.
2. **Spatial locality**: accessing data elements in near store locations.
3. **Sequential locality**: data elements are arranged and accessed linearly

Given that tree data structures are widely adopted in the video games industry (e.g., think of the KD- and of the quad-/oct-trees), we might enforce to optimize their representation both the traversal code and the tree representation [8]. We want to store trees in a cache-friendly way so that we visit the relevant information as soon as possible; please be aware that caches might work differently on different computers and architectures, and so you might get different speedups on different configurations.

We can then investigate two different possible tree representations in vector data structures: **parent-child** and **preorder** trees. *Parent-child* trees (Figure 5a) are quite common in algorithm and data structure literature [3, 8]: if we assume that each tree has a maximum

branching factor of a, then each j-th child out of a (and counting from 1) of the i-th node on the vector can be accessed using the formula $ai + j$. For well-balanced trees, these data structures provide an efficient level-wise scan and allow to save the cost of storing additional pointers for the children elements. On the other hand, such data structure might be not convenient for either performing preorder scans or non-balanced trees. In the first case, we still need to pay the cost of performing non-contiguous random memory accesses, while for the second one, we will still need to perform a $2n$ cost. In these other situations, we might represent our tree as a *preorder tree* instead (Figure 5b on the preceding page). This data structure will take more memory, as we are now required to store a boolean to check whether the left children are there, and a pointer towards the right child.

Other tree data structures, like B+Trees, are also designed to minimize the cache misses while reading from disk: we therefore refer to [6] for more information[8].

4.3 Metric Trees: Vantage Point Trees

While some data structures like KD-Trees and QuadTrees[9] might be fine for low-dimensional data, picking a proper splitting dimension for higher dimensions is challenging. For this reason, metric trees partition the space by using triangle inequalities and euclidean distances between the data points [5, 11]. Such trees are therefore very efficient to perform the following types of queries using a metric distance function δ:

- **range queries**: return all the objects r that are at most at a distance d from the point of interest.
- **k nearset neighbours query**: return all the k nearest objects to the point of interest.

The distance function in metric trees must satisfy the following metric postulates:

- **reflexivity**: $\delta(x, y) = 0 \Leftrightarrow x = y$
- **non-negativity**: $\delta(x, y) > 0 \Leftrightarrow x \neq y$
- **Symmetry**: $\delta(x, y) = \delta(y, x)$
- **triangle inequality**: $\delta(x, y) + \delta(y, z) \geq \delta(x, z)$

Reflexivity permits the zero dissimilarity just for identical objects. Non-negativity guarantees that every two distinct objects are somehow positively dissimilar. The triangle inequality is a kind of transitivity property; it says that if $\langle x, y\rangle$ and $\langle y, z\rangle$ are similar, also $\langle x, z\rangle$ are similar.

After this digression on generic metric trees, let us deep-dive into vantage point trees: VP-trees are binary trees where, for each tree and sub-tree, all the nodes appearing in the left branch are within a given distance from the root, and all the farther elements are stored in the right sub-tree (Figure 6a on the following page). The boundary between the near and the far elements is set up from a pivot element here called **vantage point**. In order to both simplify the implementation and to make the tree as balanced as possible, the vantage point is chosen from all the remaining elements to be considered as children of the current subtree.

Let us now discuss the implementation for a node of a VP-Tree: even though we're going to use a Preorder tree to represent our data structure, we're still going to store pointers to the left and right child node, here expressed as index ids within the preorder tree. Each node is parametric to the point representation, that could be any custom. Furthermore, each node will also have radius, which will be the maximum distance of the furthest node in the tree.

---

[8] See also `https://drive.google.com/file/d/0B5EQQQtU0zzpbVZ3elFsLThyRFk/view`.
[9] See [8] for more details on such data structures.

(a) Graphical representation of a `vp_node`, i.e. a node within the VP-Tree.

(b) Algorithm providing the node descent.

Figure 6 Vantage Point Trees

```
template <typename Point> struct vp_node {

   unsigned int id; ///<@ id associated to the current node
   Point pt;       ///<@ value to be compared
   double radius; ///<@ Boundary between the left and right nodes
   unsigned int leftChild = std::numeric_limits<unsigned int>::max(); ///<@ to nullptr
   unsigned int rightChild = std::numeric_limits<unsigned int>::max(); ///<@ to nullptr
   bool isLeaf = false;

   vp_node(unsigned int id, Point pt, double radius) : id(id), pt(pt), radius(radius) {}
};
```

Let us now discuss the skeletal implementation of the VP-Tree: each tree is a parametric type with respect to the data representation (i.e., `Point`) and to the `DistanceFunction` used to evaluate the distance between the points. We also assume that `DistanceFunction` should have a default constructor.

```
template <typename Point, typename DistanceFunction> struct vp_tree {
   std::vector<vp_node<Point>> tree; ///<@ representing the tree as a vector
   DistanceFunction ker;          ///<@ instance of the distance function provided by
       ↪ the template parameter
   std::mt19937 rng;              ///<@ random number generator

   // <variables for the lookup search>
```

We accept as a constructor both vectors containing some point coordinates and initializer lists[10]. In both cases, we're reserving for the tree the same size of the passed data structure, and then we're calling the insertion function reserving the elements. In order to increase the code reuse, both constructors are then going to call a common method, `recursive_restruct_tree`, which is going to complete the initialization of the tree. We're also going to implement the lookup function `topk1Search` that we're going to discuss later.

```
   /**
    * Initializing the element with a vector of elements
    * @param ls
    */
   vp_tree(const std::vector<Point>& ls) {
     tree.reserve(ls.size());
     size_t i = 0;
```

[10] See `https://en.cppreference.com/w/cpp/language/list_initialization`

```
    for (auto x : ls) tree.emplace_back(i++, x, 0);
    recursive_restruct_tree(0, ls.size());
  }

  /**
   * Initializing the tree with some static values
   * @param ls
   */
  vp_tree(std::initializer_list<Point> ls) {
    tree.reserve(ls.size());
    size_t i = 0;
    for (auto x : ls) tree.emplace_back(i++, x, 0);
    recursive_restruct_tree(0, ls.size());
  }

  /**
   * Looking for the elements near to id that are not exactely the same as him
   * @param id
   */
  void topk1Search(size_t id) {
    found.clear();
    tau   = std::numeric_limits<double>::max();
    lookUpNearsetTo(0, id);
  }

private:
  // <rest of the implementation>
}
```

For the tree insertion we're going to have two possible base cases: first (Line 3), when the node is a leaf, and when the node has only a left child that will be a leaf (Line 9): this guarantees that the two elements are near to each other. Otherwise, we have the inductive step case (starting from Line 14), where we first pick randomly a tree root using the Mersenne Twister `rng` and an uniform number distribution. Then, as per preorder tree requirement, the root of the current subtree is placed at the beginning of the current vector portion (Line 18). As we can see, all these operations can be evaluated in constant time.

Then, we calculate the vantage point as the median from the numerical distribution (Line 20) and use the `nth_element` C++ function library to put all the element that are less than the median before it while sorting those, and put all the bigger elements after the median, but without necessarily sorting it (Label **??**): this part will cost $O(n \log n)$ in the worst case scenario. Last, we're using the preorder approach to put the root of the current left subtree immediately after the current subtree root (by recursion and Line 18). Then, the algorithm is run over all the remaining partitions.

```
1   void recursive_restruct_tree(size_t first, size_t last) {
2     if (first >= last) {
3     // If the elements overlaps, then I reached a leaf node
4     tree[first].isLeaf = true;
5     tree[first].leftChild = std::numeric_limits<unsigned int>::max();
6     tree[first].rightChild = std::numeric_limits<unsigned int>::max();
7     } else {
8     if ((last - first) <= 1) {
9       // If the elements differ by two, then I decide that one is the root, and the
            ↪ other is the child
10      tree[first].radius = ker.distance(tree[first].pt, tree[last].pt);
11      tree[first].leftChild = last;
12      tree[first].rightChild = std::numeric_limits<unsigned int>::max();
13      tree[last].pt.isLeaf=true;
14    } else {
15      // Picking the root randomly as the first element of the tree ~ O(1)
16      std::uniform_int_distribution<int> uni(first, last - 1);
```

```
17      int root = uni(rng);
18      std::swap(tree[first], tree[root]);
19
20      size_t median = (first + last) / 2;
21
22      /*
23       * nth_element is a partial sorting algorithm that rearranges elements in [first,
             ↪ last) such that:
24       *
25       * - The element pointed at by median is changed to whatever element would occur
             ↪ in that position if [first, last) were sorted.
26       * - All of the elements before this new nth element are less than or equal to the
             ↪  elements after the new nth element.
27       */
28      std::nth_element(
29        tree.begin() + first + 1,//first
30        tree.begin() + median, //median
31        tree.begin() + last, //last
32        [first, this] (const vp_node<Point>& i1, const vp_node<Point>& i2) {
33          return ker.distance(tree[first].pt, i1.pt) < ker.distance(tree[first].pt, i2.
                ↪ pt);
34        });
35
36      // Setting the separating elements
37      tree[first].radius = ker.distance(tree[first].pt, tree[median].pt);
38
39      // Recursively splitting in half the elements within my radius and the ones out
40      tree[first].leftChild = first+1;
41      tree[first].rightChild = (first + last) / 2 + 1;
42      recursive_restruct_tree(tree[first].leftChild, tree[first].rightChild-1);
43      recursive_restruct_tree(tree[first].rightChild, last);
44
45    }
46    }
47  }
```

This implementation's time recurrence relation is $T(n) = 2T(\frac{n}{2}) + O(n \log n)$. By using the master theorem, we can prove that such an algorithm has a final cost of $\Theta(n \log^2 n)$. On the other hand, the original data structure had an insertion cost of $O(n \log n)$ [4, 9]. We might observe that this additional cost is the one required for representing the tree as a contiguous representation in memory. Nevertheless, this other solution guarantees that the elements are stored in a pre-visit order, thus favouring caching the nodes near to each sub-tree. Last, VP-trees might provide a quite efficient node coordinate's update function: this will make this data structure as a good candidate for collision detection algorithms over dynamic objects.

As a last operation over the VP-Tree, let us now discuss the look-up search for the elements within the VP-Tree. Let us suppose that we want to perform collision detection for a given moving object x and, for this reason, we want to get the nearest elements to x. For the tree search, we're going to use a `tau` (τ) variable to store the minimum distance found for x with any other subtree root providing the best search candidate. Given that there could be more than one candidate with the same distance τ, we're going to store those inside a vector `found`.

```
// <variables for the lookup search> :=
// Please note: this is not a thread-safe implementation
double tau;                     ///<@ current distance decision value at the iteration
std::vector<size_t> found;      ///<@ current found elements for the given lookup query
```

Now, given a root r of the current subtree, we're going to perform the following steps: get the current tree and radius information: for the current query node `id`, evaluate the

distance `dist` from the current node to r. If the node exactly matches with the current root, discard it: ee're not going to return as the best match the node itself (Line 7). Otherwise, we're going to test whether the current root has a distance less than the previous search (Line 9), so that we can discard the previous τ value with the current one and select the new candidate or, if the current distance is approximately equal to τ, continue adding the current root as another possible candidate (Line 16). Then, continue to search the best candidates recursively if possible: after remembering that the vantage point of a tree is **not** necessarily the root of the left subtree, we need to know that the point of interest `id` will be some distance from the vantage point v providing the left subtree boundary for the current root r. Therefore, we need to determine the distance between v and `id` in order to decide whether to proceed visiting the left or the right subtree.

If `id` is closer to the root than the root boundary, we search the left child first, which contains all of the points closer than the radius. We find the blue point (Figure 6b on page 16, Lines 24 and 30). Since it is farther away than tau we update the tau value. Even though we scanned all the nodes that are nearer to the radius, `id` is closer to the margin of the boundary than the farthest point that we have found. Therefore there could be closer points just outside of the boundary. We do need to descend into the right child to find the green point (Lines 27 and 33).

```
1  void lookUpNearsetTo(size_t root_id, size_t id) {
2   auto node = tree[id];
3   auto root = tree[root_id];
4   double rootRadius = tree[root_id].radius;
5   double dist = ker.distance(tree[root_id].pt, node.pt);
6
7   if ((root_id != id)) { // do not add the same object
8    // If I find a better object, then discard all the previous ones, and set the current
         ↪  one
9    if (definitelyLessThan(dist,tau)) {
10    found.clear();
11    found.emplace_back(root_id);
12    tau = dist;
13   } else
14    // Otherwise, if they are very similar, then I could add this other one too
15    if (approximatelyEqual(dist, tau)) {
16    found.emplace_back(root_id);
17    tau = std::min(dist, tau);
18   }
19  }
20
21  // Continuing with the traversal depending on the distance from the root
22  if (dist < rootRadius) {
23   if (root.leftChild != std::numeric_limits<unsigned int>::max() && dist - tau <=
         ↪ rootRadius)
24    lookUpNearsetTo(root.leftChild, id);
25
26   if (root.rightChild != std::numeric_limits<unsigned int>::max() && dist + tau >=
         ↪ rootRadius)
27    lookUpNearsetTo(root.rightChild, id);
28  } else {
29   if (root.rightChild != std::numeric_limits<unsigned int>::max() && dist + tau >=
         ↪ rootRadius)
30    lookUpNearsetTo(root.rightChild, id);
31
32   if (root.leftChild != std::numeric_limits<unsigned int>::max() && dist - tau <=
         ↪ rootRadius)
33    lookUpNearsetTo(root.leftChild, id);
34  }
35 }
```

5 Exercises

1. Assume to be dealing with self-balancing binary search trees:
   a. What is the computational complexity of inserting nodes into a binary tree?
   b. What about an insertion/search?
   c. What about removing one element?
2. What is the worst-case scenario for an unbalanced tree?
3. Do (double) linked lists preserve the principle of locality? Does 1d array preserve it instead?
4. By extending the Y operator, you can also define a memoizer for any given possible lambda function!
5. What is the best matrix multiplication for exploiting the principle of locality?
6. Change the data structure so that the choice of the subtree root is one of the nearest node to the parent.
7. What if the radius of each subtree is determined as a function of the nodes' velocity?

References

1 Andrea Asperti, Wilmer Ricciotti, and Claudio Sacerdoti Coen. Matita tutorial. *J. Formalized Reasoning*, 7(2):91–199, 2014.

2 Giacomo Bergami, André Petermann, and Danilo Montesi. Thosp: An algorithm for nesting property graphs. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, GRADES-NDA '18, 2018.

3 Alan Albert Bertossi. *Algoritmi e strutture di dati*. UTET, 2000.

4 Sergey Brin. Near neighbor search in large metric spaces. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, 1995.

5 Paolo Ciaccia, Marco Patella, Fausto Rabitti, and Pavel Zezula. Indexing metric spaces with m-tree. In *Convegno Nazionale Sistemi Evolluti per Basi di Dati, SEBD 1997, Verona, Italy, 25-26-27 Giugno 1997*.

6 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

7 Joëlle Despeyroux and André Hirschowitz. Higher-order abstract syntax with induction in coq. In *Logic Programming and Automated Reasoning, 5th International Conference, LPAR'94, Kiev, Ukraine, July 16-22, 1994, Proceedings*, pages 159–173, 1994.

8 Christer Ericson. *Real-Time Collision Detection*. CRC Press, Inc., Boca Raton, FL, USA, 2004.

9 F. Nielsen, P. Piro, and M. Barlaud. Bregman vantage point trees for efficient nearest neighbor queries. In *2009 IEEE International Conference on Multimedia and Expo*, pages 878–881, 2009.

10 Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 9th edition, 2012.

11 Tomáš Skopal and Benjamin Bustos. On nonmetric similarity search problems in complex domains. *ACM Comput. Surv.*, 43(4):34:1–34:50, October 2011.

12 Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 4th edition, 2014.

13 Chee-Keng Yap. A real elementary approach to the master recurrence and generalizations. In *TAMC*, 2011.

A Proving recursive Fibonacci's computational complexity

Given that we cannot use Lemma 4 on page 6 to prove that $T \in O(2^n)$, we need to prove it by induction. For a property P over natural numbers $n \in \mathbb{N}$, this means that we need to detect a *base case* case and and an *induction step*. The base case proves that the property holds for the numbers starting the sequence. The induction step proves that the property holds for $P(n+1)$ by assuming that the property holds for n, i.e. $P(n)$. This proof technique can be extended to all the possible well-founded data structures, and it is the basic technique used in interactive theorem provers [1, 7].

In the Fibonacci algorithm we will have two base cases, those are $n = 1$ and $n = 2$.

▶ **Lemma 7.** $T(n) \in O(2^n)$

Proof. Let us rewrite $T(n) \in O(2^n)$ as $\exists c_0, n_0.\forall n \geq n_0.T(n) \leq c_0 2^n$.

- **Base cases**: for $n = 1$ we need to choose c_0 for $T(1) = c_i \leq c_0 2^1$. This is proved for $c_0 \geq \frac{c}{2}$; similar considerations can be done for $n = 2$.
- **Inductive step**: assuming that $\forall n' < n.T(n') \leq c_0 2^{n'}$, we assert:

$$\begin{aligned} T(n) &= T(n-1) + T(n-2) \\ &\leq c_0 2^{n-1} + c_0 2^{n-2} \\ &= c_0 2^{n-2}(2+1) \\ &\leq c_0 2^{n-2} 4 \\ &= c_0 2^n \end{aligned}$$

◀