SE02 Algorithms and Data Structures Portfolio

This portfolio was created by Philip Bizimis. The main sources for the content were the books *The Algorithm Design Manual* by Steven S. Skiena, *Grokking Algorithms: An Illustrated Guide for Programmers and Other Curious People* by Aditya Bhargava, the Khanacademy course on Algorithms and the internet (more detailed sources list at the end).

This portfolio was created for the SE02 assessment but I try to build portfolios in a way that they serve me after the assessment as well. Thus, I tried to avoid long text paragraphs to give myself the chance to quickly find the notes I was looking for.

The last edit for the assessment version happened on the 13.07.2020. The pdf version will be accompanied by the same version on GitHub to interact with links.

Table of Contents

1. Algorithm Theory

> algorithm (noun): "A process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer." (Oxford Languages)

Algorithms are a part of us. We use them every day and probably do not even realize it. An algorithm can be as simple as: If I come home, I wash my hands and turn the lights on.
Nevertheless, the real power of algorithms comes in form of computer programs. There are many different applications like sorting, searching or compressing. Furthermore, there are many algorithms that, in the end, solve the same problem. To compare theses algorithms they have to be analyzed. This analyzation of an algorithm is called asymptotic analysis.

1.1 Asymptotic Analysis

Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.

There are rules to follow in order to express the correct asymptotic analysis of an algorithm:

- As the analysis is asymptotic, it refers to large inputs (n). This means that the terms that grow "most quickly" will make the other ones irrelavant.
  - If the function is a sum of several terms, the one with the largest growth rate can be kept and the others omitted.
  - If the function is a product of several factors, any constants can be omitted.

Time Complexity

The time complexity of an algorithm relates the length of an algorithm's input to the number of steps it takes.

We do not want to give the running time of an algorithm in a time unit because this would mean that it is only comparable with the same implementations (programming language, compiler, hardware, etc.). Thus, it is given as a function:

The RAM (random access machine) Model of Computation:

- this is the hypothetical computer in order to have a language- and machine-independent analysis
  - basic logical or arithmetic operations (+,*,=,if,call) are considered to be simple operations that take one time step
  - loops and subroutines are complex operations composed of multiple time steps based on the number of iterations
  - memory access takes one time step

Space Complexity

The space complexity of an algorithm relates the length of an algorithm's input to the number of storage locations it uses.

It is also expressed asymptotically. Space complexity is a measure of the amount of working storage an algorithm needs. This amount is, again, dependent on the input size. Auxiliary space is not the same as space complexity even though they are sometimes (wrongly) used for the same thing. Space complexity is the sum of the auxiliary space and the input space.

A Simple Example

```
def print_items(array):
    for item in array:
        print(item)
    return True
```

- the function runs through every item in the list once
- for an array with length n it runs n times
- constant terms (e.g. returning or printing some value) are ignored
  - (constant terms can make a difference if the compared algorithms have the same running time)
- time complexity of $\Theta(n)$

- space complexity
  - the array takes n units of space, as the length can vary

- the item variable is constant as the new item will be saved and the old item will be discarded
- thus, we have a space complexity of $\Theta(n)$
- if we talk about auxiliary space, the input array would not be counted
  - thus, we have a auxiliary space complexity of $\Theta(1)$

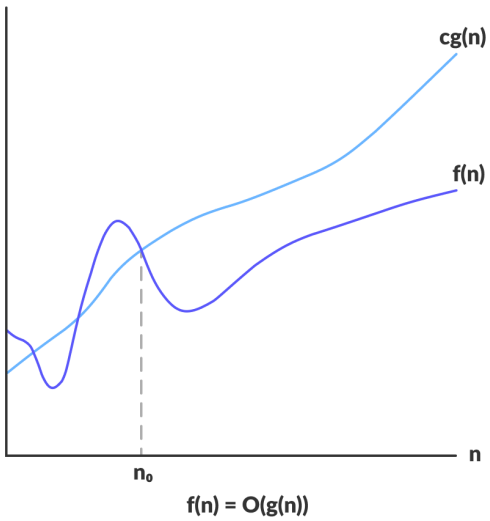
(This is a very good example why the term space complexity cannot be used for auxiliary space complexity)

1.2 Asymptotic Notation

Big-O Notation

"the running time grows at most this much, but it could grow more slowly"

- expresses the upper bound



Source (https://cdn.programiz.com/sites/tutorial2program/files/big0.png)

For a function $g(n)$, $O(g(n))$ is given by the relation:

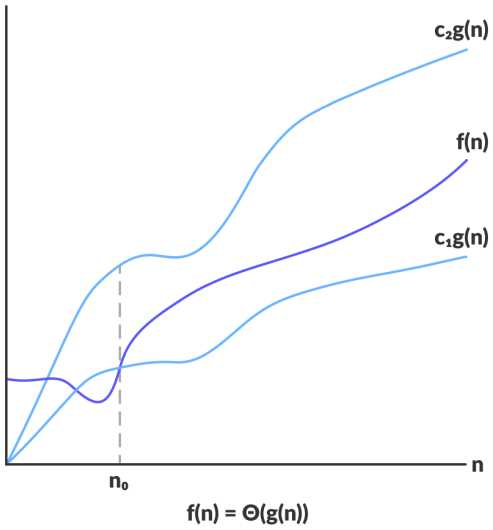
```
O(g(n)) = { f(n): there exist positive constants c and n0
            such that 0 ≤ f(n) ≤ cg(n) for all n ≥ n0 }
```

Note to self: Many use Big-O notation even if Big-Theta would be more fitting.

Big-Θ Notation

"the function grows asymptotically as fast as ,e.g., n^2 "

- expresses the upper bound and lower bound (tight bound)



Source (https://cdn.programiz.com/sites/tutorial2program/files/theta.png)

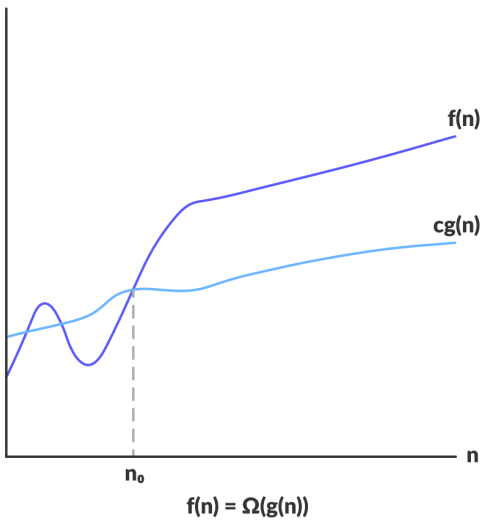
For a function $g(n)$, $\Theta(g(n))$ is given by the relation:

```
θ(g(n)) = { f(n): there exist positive constants c1, c2 and n0
            such that 0 ≤ c1g(n) ≤ f(n) ≤ c2g(n) for all n ≥ n0 }
```

Big-Ω Notation:

"the running time grows at least this much"

- expresses the lower bound



Source (https://cdn.programiz.com/sites/tutorial2program/files/omega.png)

For a function $g(n)$, $\Omega(g(n))$ is given by the relation:

```
Ω(g(n)) = { f(n): there exist positive constants c and n0
            such that 0 ≤ cg(n) ≤ f(n) for all n ≥ n0 }
```

Best, Worst, Average Case

Important to note is that for each time complexity there can be a best, worst and average case. To clarify this, I want to use this example:

```
Contrive(n)
if n = 0 then do something Theta(n^3)
else if n is even then
    flip a coin
    if heads, do something Theta(n)
    else if tails, do something Theta(n^2)
else if n is odd then
    flip a coin
    if heads, do something Theta(n^4)
    else if tails, do something Theta(n^5)
```

If this would be an algorithm, we can find different asymptotic behavior of this function.
In the best case (n is even), the runtime is $\Omega(n)$ and $O(n^2)$ but not Θ of anything.
In the worst case (n is odd), the runtime is $\Omega(n^4)$ and $O(n^5)$ but not Θ of anything.
In the case n = 0, the running time is $\Theta(n^3)$.
Source (https://cs.stackexchange.com/questions/23068/how-do-o-and-%ce%a9-relate-to-worst-and-best-case)

This also means that the instances of the size n define each case. In the example above, it is odd, even or 0. For some sorting algorithms, it can be that n is already sorted, almost sorted or sorted in reverse. For searching algorithms this may be that the looked for element is at index 0, at index n-1 or not in the input at all. Concrete examples will be given when algorithms are analyzed.

Adding Functions

- the sum of two functions is governed by the dominant one
  $O(f(n)) + O(g(n)) \rightarrow O(max(f(n), g(n)))$
  $\Omega(f(n)) + \Omega(g(n)) \rightarrow \Omega(max(f(n), g(n)))$
  $\Theta(f(n)) + \Theta(g(n)) \rightarrow \Theta(max(f(n), g(n)))$

Multiplying Functions

- as described above, constants do not affect the asymptotic behavior of a function and thus, can be omitted
  $O(c \cdot f(n)) \rightarrow O(f(n))$
  $\Omega(c \cdot f(n)) \rightarrow \Omega(f(n))$
  $\Theta(c \cdot f(n)) \rightarrow \Theta(f(n))$

- two functions in a product are important as both increase
  $O(f(n)) * O(g(n)) \rightarrow O(f(n) * g(n))$
  $\Omega(f(n)) * \Omega(g(n)) \rightarrow \Omega(f(n) * g(n))$
  $\Theta(f(n)) * \Theta(g(n)) \rightarrow \Theta(f(n) * g(n))$

Example Terms

- $O(1)$ constant
- $O(logn)$ logarithmic
- $O(n)$ linear
- $O(nlogn)$ n log n
- $O(n^2)$ quadratic
- $O(n^3)$ cubic
- $n^{O(1)}$ polynomial
- $2^{O(n)}$ exponential

Properties of Logarithm

We know the formula to change the base of a logarithm:
$log_a b = log_c b / log_c a$

This is important for asymptotic notation because, as the formula shows, it is easy to change from base-a to base-c by dividing by $log_c a$.
This conversion factor is lost to asymptotic notation whenever a and c are constants. Thus, the base of the logarithm can be omitted when analyzing algorithms.

Dominance Pecking Order

n! >> c^n >> n^3 >> n^2 >> n^{1+ε} >> n log n >> n >> √n >> log^2n >> log n >> log n/ log log n >> log log n >> α(n) >> 1

Same Time Complexity

The last thing I would like to mention is how we compare two algorithms with the same time complexity. To do that, I want to use this quote:

> *But how can we compare two Θ(n log n) algorithms to decide which is faster? How can we prove that quicksort is really quick? Unfortunately, the RAM model and Big Oh analysis provide too coarse a set of tools to make that type of distinction. When faced with algorithms of the same asymptotic complexity, implementation details and system quirks such as cache performance and memory size may well prove to be the decisive factor.*
>
> - Skiena, Steven S. 1998. The Algorithm Design Manual (page 129)

1.3 Algorithm Design Strategies

Brute Force Algorithms

- straightforward methods of solving a problem that rely on computing power and trying every possibility rather then advanced techniques to improve efficiency

Backtracking Algorithms

- systematic way to iterate through all the possible configurations of a search space
- if a part solution seems to not lead to a full solution, steps are taken back to find alternative part solutions
- either a solution is found or it can be said that there is none
- usually these algorithms are implemented recursively
- ensure correctness by enumerting all possibilities
- ensure efficiency by never visiting a state more than once

Dynamic Programming

- useful when you're trying to optimize something given a constraint (maximizing or minimizing a function)
- combines greedy algorithms that make the best local decision at each step and exhaustive search algorithms that try all possibilities and select the best always to produce the optimum result
    - systematically search all possibilites while storing results to avoid recomputing
        - efficiency and correctness
- every dynamic-programming solution involves a grid
- the values in the cells are usually what you're trying to optimize
- each cell is a subproblem: How can you divide your problem into subproblems?
- recursive subproblems that are the same over and over can look up the once computed value in the table
- optimal substructure (the optimal solution can be created from optimal solutions of its subproblems)
- overlapping subproblems (supproblem is solved multiple times)
- there is no single formula for calculating a dynamic-programming solution

Greedy Algorithms

- at each step one picks the locally optimal move
- result may not be optimal (local optimum which can be a global optimum)
- "approximate" the optimal solution

Parallel Algorithms

- parallel processing is more and more important with cloud computing, cluster computing and multicore processors
- example
    - high-resolution graphics applications must render a lot of frames per second for realistic animations
    - dividing the image into regions and assigning those to different processors might be the only solution to achieve that

Divide and Conquer

- based on recursion
- breaks a problem into subproblems that are similar to the original problem

1. Divide the problem into a number of subproblems that are smaller instances of the same problem.
2. Conquer the subproblems by solving them recursively. If they are small enough, solve the subproblems as base cases.
3. Combine the solutions to the subproblems into the solution for the original problem.

The master theorem (https://www.programiz.com/dsa/master-theorem):

The master method is a formula for solving recurrence relations of the form:
$T(n) = aT(n/b) + f(n)$,
where,
n = size of input
a = number of subproblems in the recursion
n/b = size of each subproblem. All subproblems are assumed to have the same size.
f(n) = cost of the work done outside the recursive call, which includes the cost of dividing the problem and cost of merging the solutions

Here, a ≥ 1 and b > 1 are constants, and f(n) is an asymptotically positive function.

If this is the case, the time complexity of a recursive relation is given by:

$T(n) = aT(n/b) + f(n)$

where, T(n) has the following asymptotic bounds:

1. If f(n) = $O(n^{log_b a - \epsilon})$, then T(n) = $\Theta(n^{log_b a})$
    - subproblems overshadow work to split/recombine a problem
    - the recursion tree (solution tree) is leaf-heavy
2. If f(n) = $\Theta(n^{log_b a})$, then T(n) = $\Theta(n^{log_b a} * logn)$
    - work to split/recombine a problem is comparable to subproblems
3. If f(n) = $\Omega(n^{log_b a + \epsilon})$, then T(n) = $\Theta(f(n))$
    - work to split/recombine a problem dominates subproblems
    - the recursion tree (solution tree) is root-heavy

ϵ > 0 is a constant.

Where does this formula come from? The mathematical proof can be found here (https://www.cs.cornell.edu/courses/cs3110/2012sp/lectures/lec20-master/mm-proof.pdf). More suiting to this portfolio is the intuition why this applies. A divide and conquer algorithm divides the problem into a number of subproblems. Using the variables of the formula, each subproblem is assumed to be size n/b. If we think about the solution as a tree, we get a solution tree where each node is a recursive call and the children of that node are recursive calls from that call. The leaves of the tree, as we learn in the data structures chapter, do not have children and thus, are the base cases. Each node, except of the leaf-nodes, would have a child nodes. A node does work which corresponds to the size of the subproblem n passed to that instance of the recursive call and given by $f(n)$. This is important because the time it takes to create subproblems cannot be omitted. The total amount of work is the sum of work of all nodes in the solution tree. This leads us to the formula:
$T(n) = aT(n/b) + f(n)$,
In words, this means that the running time of the algorithm ($T(n)$) is the running time of the algorithm on a, by a factor (b) reduced, subproblem ($T(n/b)$) times the amount of subproblems (a) each node has (in the recursion), added to the running time it takes to create subproblems and combine their results ($f(n)$).

The master theorem allows to covert this recurrence relation to Big-Theta form. Nevertheless, there are limitations. The master theorem cannot be used if: $T(n)$ is not monotone ($T(n) = sin(n)$)
$f(n)$ is not a polynomial ($f(n) = 2n$)
a is not a constant ($a = 2n$)
$a < 1$

Examples will be calculated for the corresponding algorithms.

---

2. Algorithm Examples

2.1 Searching

- the input is a set of n keys S, and a query key q
- the problem is: Where is q in S?

The following questions help to decide which algorithm should be used:

- How much time can you spend programming?
- Are certain items accessed more often than other ones?
- Might access frequencies change over time?
- Is the key close by?
- Is my data structure sitting on external memory?
- Can I guess where the key should be?

Chapter 14.2 in The Algorithm Design Manual

Binary Search

- efficient algorithm for finding an item from a sorted list of items
- works by repeatedly dividing in half the portion of the list that could contain the item

In [1]:
```
def binary_search(arr, low, high, x):

    if high >= low:

        guess = (high + low) // 2

        if arr[guess] == x:
            return guess
        elif arr[guess] > x:
            return binary_search(arr, low, guess - 1, x)
        else:
            return binary_search(arr, guess + 1, high, x)
    else:
        return -1

# Test array
arr = [ 2, 3, 4, 10, 40 ]
x = 40

result = binary_search(arr, 0, len(arr)-1, x)

if result != -1:
    print("Element is present at index", str(result))
else:
    print("Element is not present in array")
```

```
Element is present at index 4
```

Running time of binary search:

- array of 8
- incorrect guess: reduce the size to 4
- incorrect guess: reduce the size to 2
- incorrect guess: reduce the size to 1
- guess: either the last element is the wanted value or it is not in the array

To calculate this even faster, we can leverage the log_2. The log_2 of 8 is 3. This means that we need 3 iterations to have one element left. As seen above, the means that there is one step left: to decide if this is the wanted value or not. This means that we can find the max. iteration count with the formula $log_2(n) + 1$. For arrays with a length that is not a power of 2 we can still you this formula. For example, if n is 1000, the $log_2(1000)$ is about 9.97. Adding 1 as in the formula, we get 10.97. In the case of a decimal number, we round down to find the actual number of guesses (in this case 10).

This shows that the time complexity of binary search is $O(logn)$. This is true for both the worst and average case. Best case would be $O(1)$ if the first guess is the searched item.

Best case in Big-Omega notation would $\Omega(1)$ as well. Worst-case would be $\Omega(logn)$, as it would take at least logn many iterations to say that the item is not present.

We can see that Big-O and Big-Omega notation have the same values. This means that we can express the running time in Big-Theta notation. Best case is $\Theta(1)$ and worst and average case are $\Theta(logn)$.

Linear search

In [2]:
```
def linear_search(arr, x):

    for i in range(len(arr)):

        if arr[i] == x:
            return i

    return -1

arr = [ 2, 3, 4, 10, 40 ]
x = 3

result = linear_search(arr, x)

if result != -1:
    print("Element is present at index", str(result))
else:
    print("Element is not present in array")
```

```
Element is present at index 1
```

Looking at the linear search algorithm, we assume that this code would need the sum of one iteration times the length of the array plus the time it takes to set up the for loop and returning a value. c1 * n + c2 Where c1 is the time for one loop iteration, n the array length and c2 the time of the overhead. As learned in the asymptotic analysis chapter, we can ommit constants. We now can say that the running time grows with n. The notation is $\Theta(n)$ for average and worst case. Best case would be, of course, $\Theta(1)$.

2.2 Sorting

- the input is a set of n items
- the problem is to arrange the items in increasing/decreasing order

The following questions help to decide which algorithm should be used:

- How many keys will you be sorting?
- Will there be duplicate keys in the data?
- What do you know about your data?
  - Has the data already been partially sorted?
  - Do you know the distribution of the keys?
  - Are your keys very long or hard to compare?
  - Is the range of possible keys very small?
- Do I have to worry about disk accesses?
- How much time do you have to write and debug your routine?

Chapter 14.1 in The Algorithm Design Manual

Selection Sort

- sorts an algorithm by repeatedly finding the minimum element from the unsorted part and putting it at the beginning
- in place algorithm
  - does not save the ordered array in another memory location
  - rearranges the input array

In [3]:
```
def selection_sort(arr):
    for i in range(len(arr)):

        min_idx = i
        for j in range(i+1, len(arr)):
            if arr[min_idx] > arr[j]:
                min_idx = j

        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr

test_arr = [64, 25, 12, 22, 11]

print(selection_sort(test_arr))
```

```
[11, 12, 22, 25, 64]
```

Analyzing this algorithm:

- The outer loop runs n times $\Theta(n)$
- the inner loop runs n times then n-1 times then n-2 times ... (arithmetic series)
  - $n^2/2 + n/2$ calculates the sum of this series
  - this means that $\Theta(n^2)$ applies for the inner loop
- as learned, constant factors and lower-order terms do not matter for the asymptotic analysis
- the most significant term is $\Theta(n^2)$
  - this means that the running time of selection sort is$\Theta(n^2)$
  - we also know that selection sort always runs in $\Theta(n^2)$ time in all cases
    - this is true because the iteration amount of the loops does not depend on the order of data in the array
    - even a sorted array would lead to the same amount of iterations as an unsorted one

Insertion Sort

- builds the final sorted array one item at a time
- efficient for small data sets
- in-place algorithm

In [4]:
```
def insertionSort(arr):

    for i in range(1, len(arr)):

        key = arr[i]

        j = i-1
        while j >=0 and key < arr[j] :
                arr[j+1] = arr[j]
                j -= 1
        arr[j+1] = key


arr = [12, 11, 13, 5, 6]
insertionSort(arr)
print ("Sorted array is:")
print(arr)
```

```
Sorted array is:
[5, 6, 11, 12, 13]
```

Analyzing this algorithm:

- The outer loop runs n-1 times $\Theta(n-1)$
- the while loop runs k (the length of the subarray) times if the value that is inserted is smaller than all elements in the subarray
  - the first time: k = 1, second time: k = 2, ..., the last time: k = n-1
  - if c is the constant lines of code which are executed, the following expression calculates the sum:
    - $cn^2/2 - cn/2$
  - this means that $\Theta(n^2)$ applies for the inner loop
- there is still is the possibility that the value that is inserted is bigger than all elements in the subarray
  - this means that the loop would only run once and in total, it would run n-1 times
  - $\Theta(n)$
- special case "almost sorted"
  - every element starts at most some constant number (a) of positions from where it's supposed to be when sorted
  - at most this many steps would be needed $a * c * (n-1)$
  - this means that $\Theta(n)$ applies
  - this means that insertion sort is fast when given an almost-sorted array

summary of the time complexity:

- worst case (sorted array in reverse order): $\Theta(n^2)$
- best case (sorted array): $\Theta(n)$
- average case: $\Theta(n^2)$
- "almost sorted" case: $\Theta(n)$
- for all cases, we can notate the upper bound with: $O(n^2)$

Heapsort

If the minimum element of a min-heap data structure is removed, the hole it leaves has to be filled. This can be done by moving the element of the right-most leaf into the first position. This triggers a certain response:

- the labeling of the root may no longer satisfy the heap property
- the root should be the smallest of the three elements (root and 2 children)
- if the root is not the smallest, the smallest of the three has to be the new root
- the dissatisfied element will go down the heap until it dominates all its children or becoming a leaf node
- this will happen recursively until every node and its children have been rearranged

This down-operation is also called heapify. Exchanging the maximum element with the last element and calling heapify repeatedly gives an in-place sorting algorithm, named Heapsort.

In [5]:
```
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2

    if l < n and arr[i] < arr[l]:
        largest = l

    if r < n and arr[largest] < arr[r]:
        largest = r

    if largest != i:
        arr[i],arr[largest] = arr[largest],arr[i]


        heapify(arr, n, largest)

def heapSort(arr):
    n = len(arr)

    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)

    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)

test_array = [ 12, 11, 13, 5, 6, 7]
heapSort(test_array)
print(test_array)
```

```
[5, 6, 7, 11, 12, 13]
```

- time complexity
  - $O(nlogn)$ for all cases
    - heapify takes $O(logn)$ time and heapSort repeats this n times

---

2.3 Recursive Algorithms

"To solve a problem, solve a subproblem that is a smaller instance of the same problem, and then use the solution to that smaller instance to solve the original problem."

Recursive algorithms have:

- base cases that terminate the loop
- recursive cases that call themselves

- they can take a lot of memory due to many function calls
- improving efficiency of recursive algorithms can be achieved by
  - using tail recursion
    - the recursive call is the last thing executed by the function
    - the compiler optimizes a tail recursive function by not saving the current function's stack frame because the recursive call is the last statement and there is nothing left to do in that function
    - Python does [not support (https://stackoverflow.com/questions/13591970/does-python-optimize-tail-recursion)](https://stackoverflow.com/questions/13591970/does-python-optimize-tail-recursion) tail recursion (TCO, tail call optimization)
  - memoization (a form of caching)
    - saving results in a lookup table (e.g. python dict)
    - instead of calculating, looking it up
  - bottom-up (avoiding recursion)
    - using an iterative method to save time and space
  - memoization and bottom-up are techniques from dynamic programming

Recursive Factorial

In [6]:
```
def factorial(n):
    if n < 1:
        return 1
    else:
        return n * factorial(n - 1)

factorial(5)
```

Out[6]:
```
120
```

Tail Recursive Factorial

In [7]:
```
# example code since Python does not support tail recusion

def fact(n, a = 1):

    if (n == 0):
        return a

    return fact(n - 1, n * a)

print(fact(5))
```

```
120
```

time complexity:

- the amount of recursive calls is directly proportional to the input number
- $\Theta(n)$

Palindrome Algorithm

In [8]:
```
def palindrome(word):
    if len(word) < 2:
        return print("a palindrome")
    if word[0] == word[-1]:
        return palindrome(word[1:-1])
    else:
        return print("not a palindrome")

palindrome("rotor")
```

```
a palindrome
```

time complexity:

- recursive call will be made n/2 times, as the word is stripped of the first and last letter every call
- the amount of recursive calls is directly proportional to the input number
- $\Theta(n)$

Recursive Fibonacci

In [9]:
```
def fib(n):
    if n==1:
        return 0
    elif n==2:
        return 1
    else:
        return fib(n-1)+fib(n-2)

print(fib(8))
```

```
13
```

The recursive fibonacci has an interesting time complexity.
The calls can be expressed as F(n) = F(n-1) + F(n-2). Expressing this as a tree we get something like this:

```
0                          n
1              (n-1)                    (n-2)
2         (n-2)     (n-3)        (n-3)      (n-4)
3    (n-3)(n-4) (n-4)(n-5) (n-4)(n-5) (n-5)(n-6)
```

We are now looking for a function that satisfies these rules. We can use $a^n = a^{n-1} + a^{n-2}$. This is based of the assumption that we have a to the power of n nodes in the tree that calculate the running time of the algorithm. In the tree above, we could assume to have 2^3 nodes if the n input is 3. The problem is that the height might be n but the average number of branches is less than 2 due to the base cases. To calculate the exact value, we do:
$a^n = a^{n-1} + a^{n-2}$ divide through by a^{n-2}
$a^2 = a + 1$ Solving this with the quadratic formula we get a=1.61803 and a=-0.618303 (rounded). A negative solution would not make sense here since the running time cannot be negative. Thus, the answer is a=1.61803. This means that we can express the time complexity of this algorithm as $\Theta(1.61803^n)$.

Another interesting fact is that this value is called the "golden ratio" (https://en.wikipedia.org/wiki/Golden_ratio).

Bottom-up Fibonacci

In [10]:
```
def fib_bottom_up(n):
    a = 0
    b = 1
    if n == 1:
        return 0
    elif n == 2:
        return 1
    else:
        for i in range(2,n):
            c = a + b
            a = b
            b = c
        return c

print(fib_bottom_up(8))
```

```
13
```

We can see here that the loop runs n-2 times, thus the time complexity of the bottom-up approach is $\Theta(n)$.

Tower of Hanoi Algorithm (Multiple Recursion)

- recursion that contain a single self-reference is known as single recursion
- recursion with multiple self-references is known as multiple recusion
- another example would be the recursive fibonacci

In [11]:
```
def TowerOfHanoi(n , from_rod, to_rod, aux_rod):
    if n == 1:
        return print("Move disk 1 from rod",from_rod,"to rod",to_rod)
    TowerOfHanoi(n-1, from_rod, aux_rod, to_rod)
    print("Move disk",n,"from rod",from_rod,"to rod",to_rod)
    TowerOfHanoi(n-1, aux_rod, to_rod, from_rod)

TowerOfHanoi(5, 'A', 'C', 'B')
```

```
Move disk 1 from rod A to rod C
Move disk 2 from rod A to rod B
Move disk 1 from rod C to rod B
Move disk 3 from rod A to rod C
Move disk 1 from rod B to rod A
Move disk 2 from rod B to rod C
Move disk 1 from rod A to rod C
Move disk 4 from rod A to rod B
Move disk 1 from rod C to rod B
Move disk 2 from rod C to rod A
Move disk 1 from rod B to rod A
Move disk 3 from rod C to rod B
Move disk 1 from rod A to rod C
Move disk 2 from rod A to rod B
Move disk 1 from rod C to rod B
Move disk 5 from rod A to rod C
Move disk 1 from rod B to rod A
Move disk 2 from rod B to rod C
Move disk 1 from rod A to rod C
Move disk 3 from rod B to rod A
Move disk 1 from rod C to rod B
Move disk 2 from rod C to rod A
Move disk 1 from rod B to rod A
Move disk 4 from rod B to rod C
Move disk 1 from rod A to rod C
Move disk 2 from rod A to rod B
Move disk 1 from rod C to rod B
Move disk 3 from rod A to rod C
Move disk 1 from rod B to rod A
Move disk 2 from rod B to rod C
Move disk 1 from rod A to rod C
```

Let's calculate the time complexity of this algorithm.
The recursive equation can be expressed as:
I $T(n) = 2T(n-1) + 1$
II $T(n-1) = 2T(n-2) + 1$
III $T(n-2) = 2T(n-3) + 1$

Now we can substitute $T(n-2)$ in equation II with the term of equation III:
IV $T(n-1) = 2(2T(n-3) + 1) + 1$

Now we can substitute $T(n-1)$ in equation I with the term of equation IV:
$T(n) = 2(2(2T(n-3) + 1) + 1) + 1$
$T(n) = 2^3T(n-3) + 2^2 + 2^1 + 1$
Generalizing this term we get:
$T(n) = 2^kT(n-k) + 2^{k-1} + 2^{k-2} + \ldots + 1$
With the base case $T(0) = 1$, we get that $n - k = 0$. This allows us to put n for k:
$T(n) = 2^nT(0) + 2^{n-1} + 2^{n-2} + \ldots + 1$
The sum of this geometric progression can be expressed as:
$2^{n+1} - 1$
Ignoring the constants we get the final time complexity of $\Theta(2^n)$.

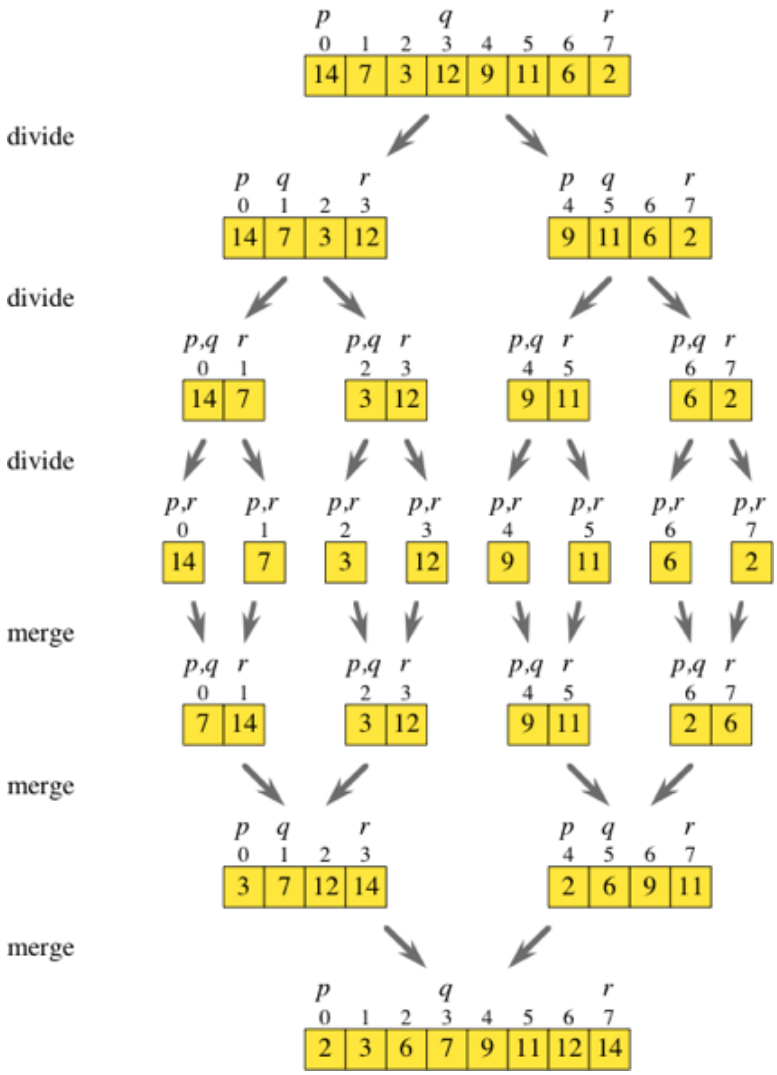
2.4 Divide and Conquer

Merge Sort

- sorting an array

p and r define the subarray

1. Divide by finding the number q of the position midway between p and r: add p and r, divide by 2, and round down.
2. Conquer by recursively sorting the subarrays in each of the two subproblems created by the divide step. That is, recursively sort the subarray array[p..q] and recursively sort the subarray array[q+1..r].
3. Combine by merging the two sorted subarrays back into the single sorted subarray array[p..r].



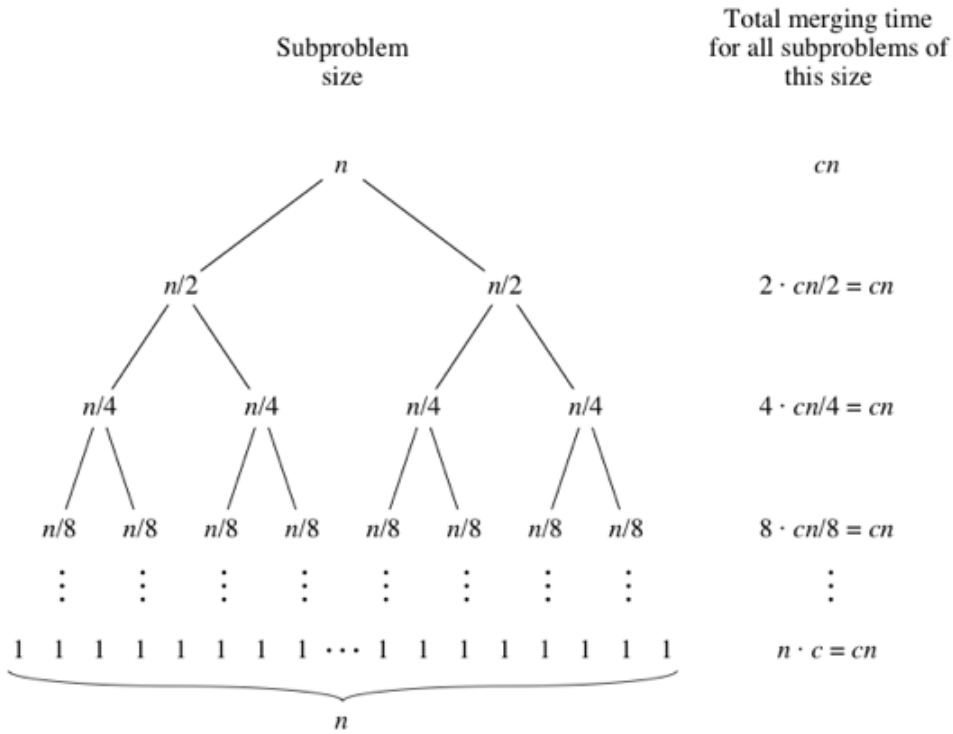
Source (https://cdn.kastatic.org/ka-perseus-images/ace963383bea8d154f6abd1322a06a73b56b4628.png)

In [12]:
```
def mergeSort(arr):
    if len(arr) > 1:
        mid = len(arr)//2
        L = arr[:mid]
        R = arr[mid:]

        mergeSort(L)
        mergeSort(R)

        i = j = k = 0

        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i+= 1
            else:
                arr[k] = R[j]
                j+= 1
            k+= 1
        while i < len(L):
            arr[k] = L[i]
            i+= 1
            k+= 1

        while j < len(R):
            arr[k] = R[j]
            j+= 1
            k+= 1

arr = [12, 11, 13, 5, 6, 7]
mergeSort(arr)
print(arr)
```

```
[5, 6, 7, 11, 12, 13]
```

Time complexity:

1. The divide step takes constant time, regardless of the subarray size. After all, the divide step just computes the midpoint q of the indices p and r. This means $\Theta(1)$.
2. The conquer step, where we recursively sort two subarrays of approximately $n/2$ elements each, takes some amount of time, but we'll account for that time when we consider the subproblems.
3. The combine step merges a total of n elements, taking $\Theta(n)$ time.



Source (https://cdn.kastatic.org/ka-perseus-images/5fcbebf66560d8fc490de2a0d8a0e5b1d65c5c54.png)

The running time is the sum of the merging times for all the levels. If there are l levels, the total merging time is $l * cn$. l can be written as $log_2 n + 1$. This means that $\Theta(nlog_2 n)$ is the running time for merge sort in Θ notation. This applies for best, worst and average case.

We can also use the master theorem to calculate the time complexity: $T(n) = aT(n/b) + f(n)$,
where,
n = size of input
a = number of subproblems in the recursion
n/b = size of each subproblem. All subproblems are assumed to have the same size.
f(n) = cost of the work done outside the recursive call, which includes the cost of dividing the problem and cost of merging the solutions

As stated in 2. we recursively sort two subproblems of the size $n/2$. This means that we have a = 2 and b = 2. n stays n and f(n) is the cost of dividing and cost of merging. As stated in 1. and 3. this is $\Theta(1)$ and $\Theta(n)$. Putting it all together the formula for merge sort becomes:
$T(n) = 2T(n/2) + \Theta(n)$,
In this case, we use the second case of the master theorem which states:

2. If f(n) = $\Theta(n^{log_b a})$, then T(n) = $\Theta(n^{log_b a} * logn)$

Let us see if this is true:
f(n) = $\Theta(n^{log_b a})$ becomes:
n = $\Theta(n^{log_2 2})$
n = $\Theta(n^1)$
n = $\Theta(n)$
This is true and thus, the time complexity is:
T(n) = $\Theta(n^{log_2 2} * logn)$
T(n) = $\Theta(n^1 * logn)$
T(n) = $\Theta(n * logn)$

Comparing this result with the previous one, we see that both are the same.

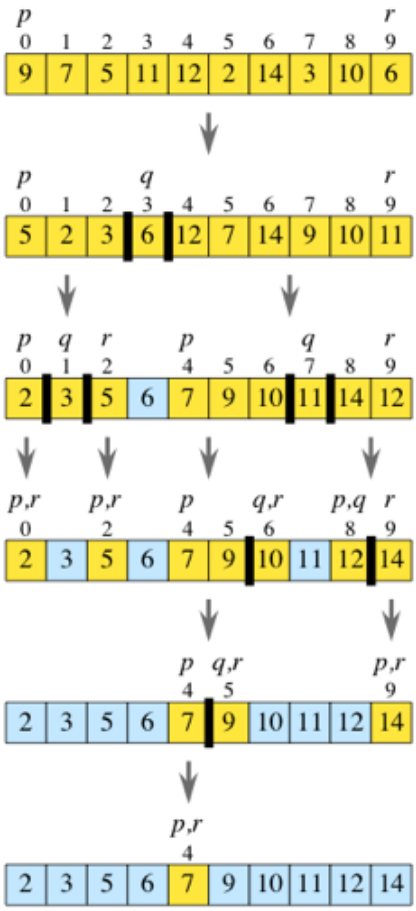
Also to note:

- during merging, two copies of the entire array are made (L and R)
- because it copies more than a constant number of elements at some time, merge sort does not work in place
- if space is limited, in-place algorithms are preferred

Quicksort

- sorting an array
- in place algorithm

1. Divide by choosing any element in the subarray array [p..r] (the pivot). Rearrange the elements in array [p..r] so that all elements in array [p..r] that are less than or equal to the pivot are to its left and all elements that are greater than the pivot are to its right (partitioning).
2. Conquer by recursively sorting the subarrays array [p..q-1] and array [q+1..r].
3. Combine by doing nothing. The elements in array[p..r] can't help but be sorted!



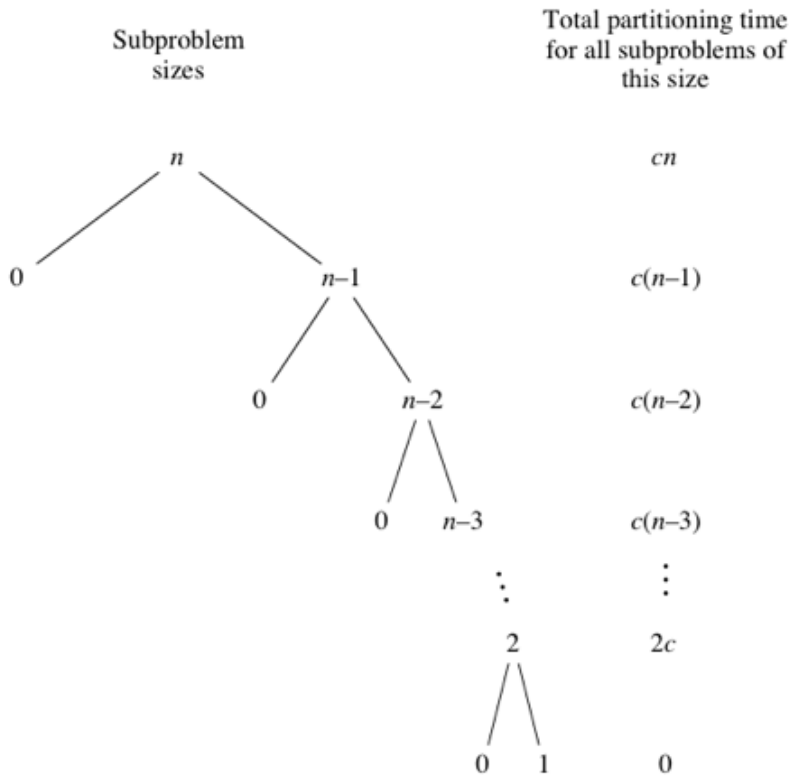
Source (https://cdn.kastatic.org/ka-perseus-images/9876d4dc59e01a4742860ae1831c20f654ed7959.png)

In [13]:
```
def partition(arr,low,high):
    i = ( low-1 )
    pivot = arr[high]

    for j in range(low , high):
        if   arr[j] <= pivot:

            i = i+1
            arr[i],arr[j] = arr[j],arr[i]

    arr[i+1],arr[high] = arr[high],arr[i+1]
    return ( i+1 )

def quickSort(arr,low,high):
    if low < high:
        q = partition(arr,low,high)
        quickSort(arr, low, q-1)
        quickSort(arr, q+1, high)

arr = [10, 7, 8, 9, 21, 50]
n = len(arr)
quickSort(arr,0,n-1)
print ("Sorted array is:")
print(arr)
```

```
Sorted array is:
[7, 8, 9, 10, 21, 50]
```

worst case running time:

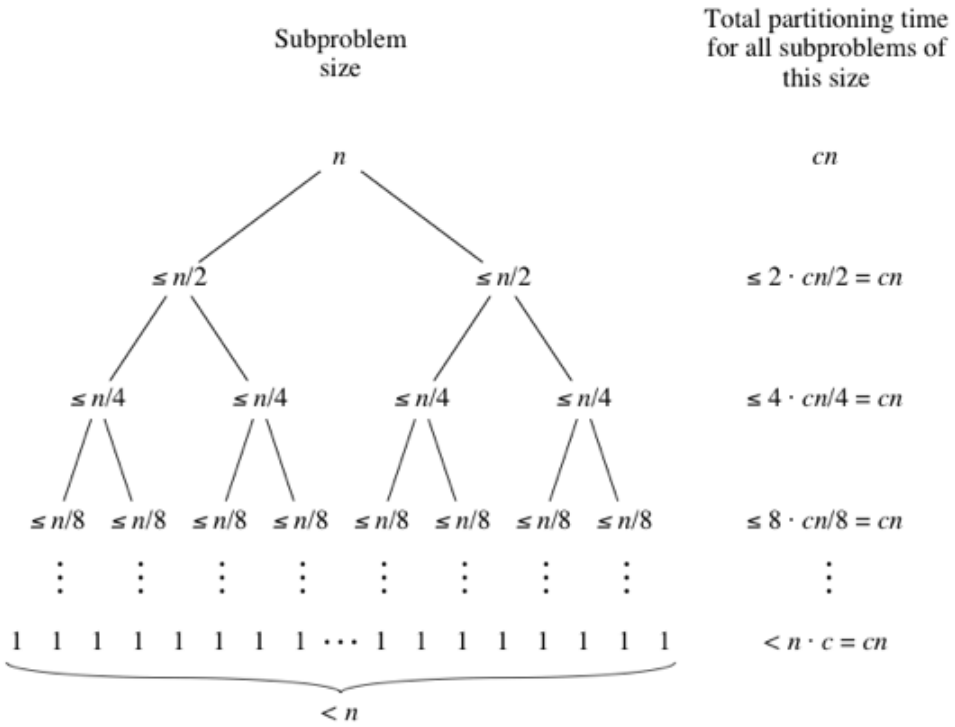
- partitions are the most unbalanced $\Theta(n^2)$



Source (https://cdn.kastatic.org/ka-perseus-images/7da2ac32779bef669a6f05decb62f219a9132158.png)

best case running time:

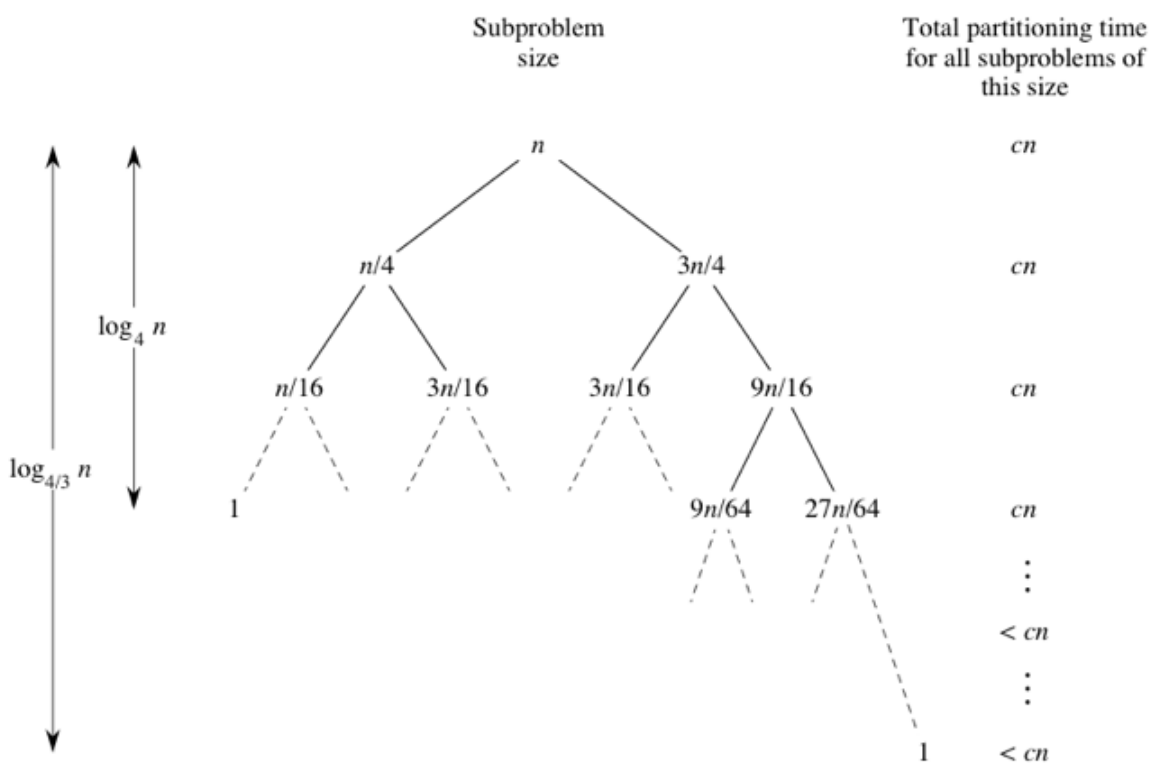
- partitions are evenly balanced (same or difference of 1) $\Theta(nlogn)$



Source (https://cdn.kastatic.org/ka-perseus-images/21cd0d70813845d67fbb11496458214f90ad7cb8.png)

average case running time:

- math explained here (https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort) $O(nlogn)$



Source (https://cdn.kastatic.org/ka-perseus-images/130b2d2a1fe897253def054f4c3aa7bd94cb6cf2.png)

We can also use the master theorem to calculate the time complexity: $T(n) = aT(n/b) + f(n)$,
where,
n = size of input
a = number of subproblems in the recursion
n/b = size of each subproblem. All subproblems are assumed to have the same size.
f(n) = cost of the work done outside the recursive call, which includes the cost of dividing the problem and cost of merging the solutions

We can see that the number of subproblems in the recursion is of the size 2. Furthermore, we get subproblems with a size of $n/2$ since the pivot rearranges the array with respect to bigger and smaller values. The cost of the work that is done outside of the recursive calls, which is partitioning the array, has a time complexity of $\Theta(n)$. This leaves us with the variables a=2, b=2 and f(n)=n.
$T(n) = 2T(n/2) + \Theta(n)$,
These are the same values as for the merge sort algorithm. Thus, we can say that we have a time complexity of:
T(n) = $\Theta(n * logn)$

Nevertheless, there is a difference. The recurrence above applies for the best case. For the worst case, we have a different recurrence:
As we can see in the worst case tree graphic above, in this case we choose a pivot that returns the most unbalanced partitions. This means that we only have one recursive call and the subproblems are of the size n-1. The partitioning function still has $\Theta(n)$.
$T(n) = T(n-1) + \Theta(n)$,

The master theorem is designed for the case where

1. the number of subproblems is a constant, and
2. the sizes of the subproblems decay geometrically

The recurrence for the worst case has the problem size decay linearly and thus, the second requirement does not hold. This means, that we cannot solve this recurrence by using the master theorem. We can still unroll the recurrence ourselves to solve it:
$\Theta(n + (n-1) + (n-2) + \ldots + 2 + 1)$
$\Theta(n(n+1)/2)$
$\Theta(n^2)$

Comparing this result with the previous one, we see that both are the same.

Last, for the average case we get the recurrence:
$T(n) = 1/n * sum[T(i) + T(n-i-1), i = 0 \ldots n-1] + \Theta(n)$,
This again cannot be solved by the master theorem but by unrolling again. The term can be summarized by:

$O(n * logn)$.
The exact math can be found here (http://www.hananayad.com/teaching/syde423/quickSortAvgCase.pdf)

2.5 Breadth-First Search

General

- example application: path finding
- visits each connected node of the current node
  - 1st degree neighbors first
  - 2nd degree neighbors second
  - ...
  - thus, finds the nearest goal neighbor (shortest path)
- terminates when it finds a/the goal node
- keeps track of nodes that have already been visited but have not yet been visited from
  - queue (first in, first out)

Analysis of BFS

- graph with Node set N and Edge set E
- running time is $O(V + E)$:
  - since every node and edge will be explored in the worst case
  - $O(E)$ may very between $O(1)$ and $O(V^2)$ (adjacency matrix), depending on what the input graph is

meaning of $O(V + E)$

- for $|E| \geq |V|$
  - $|E| + |V| \leq |E| + |E| = 2 * |E|$
  - ignoring constant factors in asymptotic notation, we get $O(E)$
- for $|E| < |V|$
  - $|V| + |E| \leq |V| + |V| = 2 * |V|$
  - ignoring constant factors in asymptotic notation, we get $O(V)$
- this means that $(O(V + E)$ really means $O(max(V, E))$

In [14]:

```
actions=(
    "plug_in_toaster",
    "unplug_toaster",
    "put_in_bread",
    "take_out_bread",
    "switch_on_toaster",
    "wait")

state={
    "toaster_has_power":False,
    "toaster_is_on":False,
    "bread_location":"plate",
    "bread_state":"untoasted",
    "time":0
    }

def goal(state):
    return state["bread_location"] == "plate" and state["bread_state"] == "toasted"

def state_transition(state, action):
    newState = state.copy()
    if action =="plug_in_toaster":
        # toaster now has power
        newState["toaster_has_power"] = True
        newState["time"] += 1
    elif action =="unplug_toaster":
        # unpower toaster and stop toasting process
        newState["toaster_has_power"] = False
        newState["toaster_is_on"] = False
        newState["time"] += 1
    elif action == "put_in_bread":
        # move bread into toaster. Only possible if toaster is not on (casue it locks)
        if not newState["toaster_is_on"]:
            newState["bread_location"] = "toaster"
        newState["time"] += 1
    elif action == "take_out_bread":
        # move bread from toaster to plate. Only possible if toaster is not on (casue it locks)
        if not newState["toaster_is_on"]:
            newState["bread_location"]="plate"
        newState["time"] += 1
    elif action == "switch_on_toaster":
        # switch on the toaster
        if newState["toaster_has_power"]:
            newState["toaster_is_on"]=True
        newState["time"] += 1
    elif action == "wait":
        # wait for ten steps
        newState["time"] += 10
        # if toaster is on, it is switched off, if bread was in toaster, it is toasted now.
        if newState["toaster_is_on"]:
            if newState["bread_location"] == "toaster":
                newState["bread_state"] = "toasted"
            newState["toaster_is_on"] = False
    return newState


# BFS algorithm
def plan(start_state):

    to_visit = [(start_state, [])]
    loop_runs = 0
    counter = 0

    while(to_visit):
        loop_runs += 1
        counter += 1

        (state, path) = to_visit.pop(0)

        if goal(state):
            print("BFS needed " + str(loop_runs) + " loop iterations.")
            return path
```

```
        for action in actions:
            to_visit.append((state_transition(state, action), path + [action]))


def test(start_state):
    print("testing:",start_state)

    # call plan function
    #sequence = plan(start_state)
    sequence = plan(start_state)
    print("found sequence:",sequence)

    # apply plan to start state
    state = start_state
    for action in sequence:
        state = state_transition(state,action)
    # check whether result fulfills the goal
    print("fulfills goal?", goal(state))
    print("in world time",state["time"])

# execute the test for a few test cases
test(state)
# test({'toaster_has_power': True, 'toaster_is_on': False, 'bread_location': 'toaster', 'bread_stat
# test({'toaster_has_power': True, 'toaster_is_on': True, 'bread_location': 'plate', 'bread_state':
```

```
testing: {'toaster_has_power': False, 'toaster_is_on': False, 'bread_location': 'plate', 'bread_st
ate': 'untoasted', 'time': 0}
BFS needed 2165 loop iterations.
found sequence: ['plug_in_toaster', 'put_in_bread', 'switch_on_toaster', 'wait', 'take_out_bread']
fulfills goal? True
in world time 14
```

This is a planning AI used to find the right action sequence to toast bread. It was a homework task for the AI module. In this case, the graph nodes are different states. The BFS algorithm finds the nearest goal statee. Due to the time parameter, this might be the nearest state but not in regards to the time of the "toaster world" (graph is weighted but for this example the weights can be ignored)

State neighbors (new nodes) are found by applying the state transition function on the current state (current node). The resulting state, is the neighbor of the previous state and thus, can be added to the queue.

2.6 Depth-First Search

General

- example: path finding
- explores as far as possible along each subtree before backtracking
- terminates when it finds a goal node
- keeping track of nodes that have already been visited but have not yet been visited from
    - stack (last in, first out)

Analysis of DFS

- running time is the same as BFS
    - a difference that DFS may never terminate (see example below)
    - BFS will always find a solution (may take a lot of time), DFS might not

In [15]:
```
# DFS algorithm
def plan(start_state):

    to_visit = [(start_state, [])]
    loop_runs = 0
    counter = 0

    while(to_visit):
        loop_runs += 1
        counter += 1

        (state, path) = to_visit.pop(0)

        if goal(state):
            print("BFS needed " + str(loop_runs) + " loop iterations.")
            return path

        for action in actions:
            to_visit.insert(0, (state_transition(state, action), path + [action]))
```

Above, we can see a code snippet of DFS that could be used for the AI task of the BFS example. The problem is that the tree structure of the state space is continuously growing as we apply the state transition function on our current state (node). Thus, we need to hope that either the state transition function cannot be applied at some point or that the goal state is found. Otherwise, this search will never terminate.

The only difference between DFS and BFS is the adding of new nodes to the to_visit data structure.

```
to_visit.append((state_transition(state, action), path + [action]))
# first in, first out logic (queue)

to_visit.insert(0, (state_transition(state, action), path + [action]))
# last in, first out logic (stack)
```

(There is better solution for Python queues which are of the `collections.deque` class but I found the solution above shows the difference more clearly.)

2.7 Greedy Algorithms

- at each step one picks the locally optimal move
- faster but not optimal (local optimum which can be a global optimum)
- optimization

approximation algorithms are judged by:

- how fast they are
- how close they are to the optimal solution

Example Algorithm

You start a radio show and want to reach as many people in all 50 states as possible. You have to decide what stations to play on to reach many people. It costs money to be on each startion. How to reach as many people and spend the least amount of money?

1. List every possible subset of stations.
   - this is calculated by 2^n, where n is 50
2. Pick the set with the smallest number of stations that covers all 50 states.

This means that it takes $O(2^n)$ time. For a large n this can take a long time to be solved. This is where greedy (approximation) algorithms are useful.

1. Pick the station that covers the most states that haven't been covered yet. Overlap does not matter.
2. Repeat until all states are covered.

In this case the greedy algorithms runs in $O(n^2)$ time.

This type of problem is called NP-complete problem:

- nondeterministic polynomial complete problem
- algorithms will not solve these problems quickly
- see radio problem above for a large n
- if you deal with a NP-complete problem
  - solve it using approximation
- it is not easy to define NP-completeness

In [16]:
```
# example greedy algorithm

states_needed = set(["mt", "wa", "or", "id", "nv", "ut", "ca", "az"])

stations = {}
stations["kone"] = set(["id", "nv", "ut"])
stations["ktwo"] = set(["wa", "id", "mt"])
stations["kthree"] = set(["or", "nv", "ca"])
stations["kfour"] = set(["nv", "ut"])
stations["kfive"] = set(["ca", "az"])

final_stations = set()

while states_needed:
    best_station = None
    states_covered = set()
    for station, states_for_station in stations.items():
        covered = states_needed & states_for_station
        if len(covered) > len(states_covered):
            best_station = station
            states_covered = covered

    states_needed -= states_covered
    final_stations.add(best_station)

print(final_stations)
```

```
{'ktwo', 'kone', 'kthree', 'kfive'}
```

Dijkstra's Algorithm

- works with weighted graphs
  - weights are positive
  - for negative weights use Bellman-Ford algorithm
- finds the path with the smallest total weight
- only works with directed acyclic graphs

How it works:

1. Finds the "cheapest" node.
2. Update costs of neighboring nodes
3. Repeat for every node
4. Calculate final path

In [17]:

```
# the graph
graph = {}
graph["start"] = {}
graph["start"]["a"] = 6
graph["start"]["b"] = 2

graph["a"] = {}
graph["a"]["fin"] = 1

graph["b"] = {}
graph["b"]["a"] = 3
graph["b"]["fin"] = 5

graph["fin"] = {}


infinity = float("inf")
costs = {}
costs["a"] = 6
costs["b"] = 2
costs["fin"] = infinity


parents = {}
parents["a"] = "start"
parents["b"] = "start"
parents["fin"] = None

processed = []

def find_lowest_cost_node(costs):
    lowest_cost = float("inf")
    lowest_cost_node = None

    for node in costs:
        cost = costs[node]

        if cost < lowest_cost and node not in processed:

            lowest_cost = cost
            lowest_cost_node = node
    return lowest_cost_node


node = find_lowest_cost_node(costs)

while node is not None:
    cost = costs[node]

    neighbors = graph[node]
    for n in neighbors.keys():
        new_cost = cost + neighbors[n]

        if costs[n] > new_cost:

            costs[n] = new_cost

            parents[n] = node

    processed.append(node)

    node = find_lowest_cost_node(costs)

print("Cost from the start to each node:")
print(costs)
```

```
Cost from the start to each node:
{'a': 5, 'b': 2, 'fin': 6}
```

Time complexity:

The general time complexity of the Dijkstra's algorithm for any data structure of the vertex set Q is:
$\Theta(|E| * T_{dk} + |V| * T_{em})$
T_{dk} and T_{em} represent the complexities of the decrease-key and extract-minimum operations in Q.

As this statement shows, the time complexity is quiet sensitive to data structures that are used to implement the algorithm. Furthermore, the difference between a adjacenty list or matrix have an impact on the running time of the Dijkstra's algorithm. I think that these sources (1 (https://stackoverflow.com/questions/26547816/understanding-time-complexity-calculation-for-dijkstra-algorithm), 2 (https://www.geeksforgeeks.org/dijkstras-algorithm-for-adjacency-list-representation-greedy-algo-8/#:~:text=We%20have%20discussed%20Dijkstra's%20algorithm,O(V%5E2)), 3 (https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#Running_time)) explain the calculations very well.

2.8 ASTAR (A*) Algorithm

- can be seen as an extension of the Dijkstra's algorithm
- is a backtracking algorithm
- uses heuristics to achieve better perforance

- the algorithm uses three functions for each node n
  - $g(n)$: the cost from the start node to n
  - $h(n)$ (heuristic function): estimates the cost of the cheapest path from n to the goal
  - $f(n) = g(n) + h(n)$
- A* selects the path that minimizes the $f(n)$

- the A* algorithm is
  - complete (if there is a solution, it will be found)
  - optimal (the optimal solution will be found, if there are more than one, one of them, dependent on the implementation, will be found)
  - optimal efficient (there is no algorithm that will find the solution faster with the same heuristic)

- the time complexity only has low importance as the strength is the heuristic search and most nodes will not be visited
- as an example, the time complexity can be meaningful for path finding tasks in a maze as A* struggels with the heuristic search
  - there have to be simplified assumptions
    - monotone heuristic: estimate is always less than or equal to the estimated distance from any neighbouring vertex to the goal, plus the cost of reaching that neighbour
    - implementation of the open and closed list have to be efficient data structures
      - open list as a binary heap, closed list as an array
- with the these assumptions, the worst-case running time is $O(V^2)$

- more generally the time complexity is noted as $O(b^d)$
  - b is the branching factor of the tree (average number of successors per state)
  - d is the depth of the goal node
- one drawback is the space complexity
  - all generated nodes are saved in memory (open and closed list)
  - this means that the worst-case has a space complexity of $O(b^d)$

The code below was a homework for the AI guild. This is my solution and there was no general solution. This means, that this might not be the optimal heuristic. Usually for path finding, the values for every node are assigned as stated above. This was not possible since the tree structure we get from the state space does not have a "distance" that can be calculated to find the heuristic value. I tried to solve this problem by doing the following:

1. The g-value of the new node is the time-factor of the current state
2. The heuristic is the estimated "distance" from the current state to the goal state. To achieve that, I assigned a counter to the node if the state come closer to the goal state. I could not just look at the state and compare it because some states changed from False to True to False in order to change other states. This meant, that I needed to know if the state is currently in the first or second False state. To do that, I used logic to reason and assign a value:

```
#testing and keeping track if actions that need multiple times (false, true, false)
if action == "put_in_bread" and current_node.state["bread_location"] == "plate" and node_s
tate["bread_location"] == "toaster":
    cbreadloc += 1

if action == "take_out_bread" and current_node.state["bread_location"] == "toaster" and cu
rrent_node.state["toaster_is_on"] == False and current_node.state["bread_state"] == "toast
ed":
    cbreadloc += 2

if action == "switch_on_toaster" and node_state["toaster_is_on"] == True and current_node.
state["toaster_is_on"] == False:
    cswitch += 1

if action == "wait" and current_node.state["toaster_has_power"] == True and current_node.s
tate["toaster_is_on"] == True:
    cswitch += 2
```

After that, I was able to assign the right heuristic value to the node.

3. The last step was to calculate the total node cost. I noticed that if I do $f(n) = g(n) + h(n)$, the node with the right action will be punished. This is because the wait action, which is necessary to complete the algorithm, takes 10 time units. All other actions only use 1 time unit. This means that the state with wait action has a higher g-value than most of the other states' g-values. Thus, the node is not selected until it's total cost is lower than all the nodes in the the open list. To solve this, I calculated the total node cost as $f(n) = g(n) - h(n)$. With the reasoning above, the counter indicated what heuristic value has to be assigned to balance out the wait action time punishment.

```
if key == "toaster_is_on":
 if current_node.cswitch == 1:
     h_count += 1
 elif current_node.cswitch > 1:
     h_count += 9
elif key == "bread_location":
 if current_node.cbreadloc == 1:9
     h_count += 1
 elif current_node.cbreadloc > 1:
     h_count += 3
elif end_state[key] == value:
 h_count += 2
```

In action, this algorithm takes way less loop iterations than the BFS. I tested different starting states and here are some numbers:
BFS - A*
2165 - 7
221 - 4
8645 - 52 (here BFS found a sequence that takes 23 time units and A* found one with 15)

To show some self reflection, I think that the heurisic and the calculation of the total cost are probably not optimal. Furthmore, I tested different values for the heuristic and found that the current ones work the best. Nevertheless, I am happy that I was able to implement the A* algorithm on my own and solved the problems I faced while doing so.

In [18]:

```
class Node():

    def __init__(self, parent=None, state=None, action=None, cswitch=0, cbreadloc=0):
        self.parent = parent
        self.state = state
        self.action = action
        self.cswitch = cswitch
        self.cbreadloc = cbreadloc

        self.g = 0 #distance between current node and start node
        self.h = 0 #heuristic, estimated distance from the current node to the end node
        self.f = 0 #total node cost

    def __eq__(self, other):
        return self.state == other.state

def calculate_h(current_node, end_node):
    current_state = current_node.state
    end_state = end_node.state
    h_count = 0

    for key, value in current_state.items():
        if key == "time":
            continue

        if key == "toaster_is_on":
            if current_node.cswitch == 1:
                h_count += 1
            elif current_node.cswitch > 1:
                h_count += 9
        elif key == "bread_location":
            if current_node.cbreadloc == 1:
                h_count += 1
            elif current_node.cbreadloc > 1:
                h_count += 3
        elif end_state[key] == value:
            h_count += 2

    return h_count


def astar(start):

    end = {'toaster_has_power': True, 'toaster_is_on': False, 'bread_location': 'plate', 'bread_sta

    start_node = Node(None, start, None)
    start_node.g = start_node.h = start_node.f = 0
    end_node = Node(None, end, None)
    end_node.g = end_node.h = end_node.f = 0

    open_list = []
    closed_list = []

    open_list.append(start_node)

    loop_counter = 0
    while(open_list):
        loop_counter += 1

        current_node = open_list[0]
        current_index = 0
        for index, item in enumerate(open_list):
            if item.f < current_node.f:
                current_node = item
                current_index = index

        # print(current_node.state)
        # print(current_node.f)

        open_list.pop(current_index)
        closed_list.append(current_node)
```

```
        if goal(current_node.state):
            path = []
            current = current_node
            print("ASTAR needed " + str(loop_counter) + " rounds.")
            while current.action is not None:
                path.append(current.action)
                current = current.parent
            return path[::-1]

        children = []

        for action in actions:
            node_state = state_transition(current_node.state, action)

            cbreadloc = current_node.cbreadloc
            cswitch = current_node.cswitch

            #testing and keeping track if actions that need multiple times (false, true, false)
            if action == "put_in_bread" and current_node.state["bread_location"] == "plate" and nod
                cbreadloc += 1

            if action == "take_out_bread" and current_node.state["bread_location"] == "toaster" and
                cbreadloc += 2

            if action == "switch_on_toaster" and node_state["toaster_is_on"] == True and current_no
                cswitch += 1

            if action == "wait" and current_node.state["toaster_has_power"] == True and current_nod
                cswitch += 2

            # Create new node
            new_node = Node(current_node, node_state, action, cswitch, cbreadloc)

            # Append
            children.append(new_node)

        # Loop through children
        for child in children:

            # Child is on the closed list
            if child in closed_list:
                continue

            # Create the f, g, and h values
            child.g = child.state["time"]
            child.h = calculate_h(child, end_node)
            child.f = child.g - child.h

            # print(loop_counter)
            # print(child.action)
            # print(child.state)
            # print(child.f)
            # print(" ")

            # Add the child to the open list
            open_list.append(child)
```

2.9 Nearest Neighbour Search Algorithm

(This was written after the data structure explanation, please read this after the kd-tree paragraph in the data structures chapter)

One of the most important operation on the kd-tree is the nearest neighbour search. This algorithm aims to find the point in the tree that is nearest to a given input point.

- using the example kd-tree from the data structure paragraph:

    - starting at the root node, as the depth is 0, the x-axis is the split dimension
        - the input point x-value would be compared with the root x-value and either go left or right depending on if the value is greater or lesser than the root x-value
        - this will be done for every depth with the change of the split dimension (x-axis, y-axis, x-axis ...)
        - the current node will become current best if the node is nearer than the current node
        - the algorithm checks if there are nodes in the circle (the radius is the distance between the input point and the current best node and the input point is the middle point)
            - this has to be done because points can be in the other half-space but be nearer
            - in the code example below, this is not done by a circle but by comparing the distance between the current best point and the input point with the distance between the input point and the cutting axis
                - if the distance of the current best and the input point is greater, the other plane could contain points that are nearer, the algorithm is then executed for the other half-space
        - the algorithm terminates once the current best node becomes the best node (no better or new nodes)

- a real life example:
    - If you ask google maps for, e.g., police stations. The algorithm can quickly find you the nearest police station to your location.

- issues:
    - if data set contains only a small number of points (n<100) other search algorithms can be more efficient (e.g. linear search)
    - as stated in the data structure paragraph, high-dimensional kd-trees should be avoided and this means that nearest neighbor search gets progressively harder as the dimensionality increases
    - static or dynamic data set
        - either the kd-tree can be build every search if the data set has only occasional insertions
        - if they are frequent, the kd-tree should support insertions and deletions

This (https://www.youtube.com/channel/UCEbYhDd6c6vngsF5PQpFVWg) YouTube channel helped me a lot to understand the theory and code.

In [19]:

```python
# build a kd-tree
def build_kdtree(points, depth=0):

    if not points:
        return None

    k = len(points[0]) # assume all points have the same dimension

    axis = depth % k # explained in data structure paragraph

    sorted_points = sorted(points, key=lambda point: point[axis])

    median = len(points) // 2

    return {
        'point': sorted_points[median],
        'left': build_kdtree(sorted_points[:median], depth + 1),
        'right': build_kdtree(sorted_points[median+1:], depth + 1)
    }


# nearest neighbour search algorithm
def distance_squared(point1, point2):
    # assume 2-dimensional points
    x1, y1 = point1
    x2, y2 = point2

    dx = x1 - x2
    dy = y1 - y2

    return dx * dx + dy * dy

def closest_point(all_points, new_point):
    best_point = None
    best_distance = None

    for current_point in all_points:
        current_distance = distance_squared(new_point, current_point)

        if best_distance is None or current_distance < best_distance:
            best_distance = current_distance
            best_point = current_point

    return best_point

def closer_distance(pivot, p1, p2):
    if p1 is None:
        return p2

    if p2 is None:
        return p1

    d1 = distance_squared(pivot, p1)
    d2 = distance_squared(pivot, p2)

    if d1 < d2:
        return p1
    else:
        return p2

def kdtree_closest_point(root, point, depth=0):
    if root is None:
        return None


    k = len(root["point"])

    axis = depth % k

    next_branch = None
    opposite_branch = None
```

```
    # checking for the current axis if the point has a greater or lesser "coordinate" value
    # then setting the next_branch and opposite_branch to the subtree of the root node

    if point[axis] < root['point'][axis]:
        next_branch = root['left']
        opposite_branch = root['right']
    else:
        next_branch = root['right']
        opposite_branch = root['left']


    # traverse the tree for the best point and check (closer_distance) if the new point has a close
    best = closer_distance(point, kdtree_closest_point(next_branch, point, depth + 1), root['point'

    # this checks for a better (nearer) point on the opposite branch (as described in the introduct
    # it does this by comparing the distance of the current best and the point with the distance be
    if distance_squared(point, best) > (point[axis] - root['point'][axis]) ** 2:
        best = closer_distance(point,
                               kdtree_closest_point(opposite_branch,
                                                    point,
                                                    depth + 1),
                               best)

    return best



data_set = [(2,3), (5,4), (9,6), (4,7), (8,1), (7,2)]

kd_tree = build_kdtree(data_set)
kdtree_closest_point(kd_tree, (5,9))
```

Out[19]: (4, 7)

Time complexity of kd-tree construction:

- the kd-tree depth is $O(logn)$
- sorting and finding the median takes a certain amount of time depending on the chosen sorting algorithm
  - $O(nlogn)$ if Heapsort or Mergesort are used
  - in our case, the sorted() function takes $O(nlogn)$ time (Timsort)
- this happens at every level
- thus, the time complexity of the construction in the example above is $O(nlog^2n)$

Time complexity of nearest neighbour search using kd-trees:

- as we calculated above the depth of the kd-tree is $O(logn)$
- in the average case kNN traverses through the levels of the tree and therefore has a time complexity of $O(logn)$
- in the worst case, there are a maximum amount of backtracking and traversing which means a worst case time complexity of $O(n)$

---

3. Data Structures

Data structures are structures that save and organize data. Different data structures have different attributes and using the right one makes data handling more efficient. Besides the data they hold, each data structure has certain operations that can be performed. Data structures are essential to every program. It is important to know different data structures and their theory to choose the right one(s) in order to have an efficient application.

I tried not to use code examples since the theory is language independent and I tried to understand data structures in that independent way. Furthermore, there are a lot of code examples for different languages in the internet and in the books I read.

> However, in practice, it is more important to avoid using a bad data structure than to identify the single best option available.
>
> - Skiena, Steven S. 1998. The Algorithm Design Manual (page 367)

3.1 Linked List

- a linear data structure
- dynamic size
- collection of items not stored at contiguous memory locations
- each item stores the address (pointer) of the next item in the list
- "like a treasure hunt"
- homogeneous or heterogeneous data
- problem: give me the last item (has to go to all the nodes to find the last one)
- reading time: $O(n)$
- insertion/deletion time: $O(1)$
  - changing the address of previous and following element
- in contrast to arrays, there is only an overflow if the memory is full
- disadvantage:
  - require extra space for storing pointer fields
  - can only do sequential access
  - arrays allow better memory locality and chache performance than random pointer jumping
- doubly linked lists
  - contain an extra pointer to the previous element
  - thus, can be traversed in both directions
  - insertion before a given node is more efficient
  - disadvantage
    - extra space for the pointer
    - extra modification of a pointer when operations are done

3.2 Array

- linear data structure
- collection of items stored at contiguous memory locations (fixed size)
- homogeneous data
- indexing is possible
- reading time: $O(1)$
- insertion/deletion time: $O(n)$
  - shifting all other elements in memory
- they are space efficient
  - consist purely of data (no links or other formatting information)
- the size of an array cannot be adjusted in the middle of a program's execution
  - the solution is a dynamic array
    - doubling the size of the array from m to $2m$ each time it runs out of space
    - copying the contents from the old array to the new one
- sequential and random access possible

3.3 Stack

- linear data structure
- insert to the top of the stack (push)
- read and take it from the top of the stack (pop)
- last in, first out or first in, last out
- push, pop time: $O(1)$ (best case implementation)
- example:
  - recursive function call:
    - first function is called
      - stack: memoryfunction1
    - first function calls second function
      - stack: memoryfunction1, memoryfunction2

        - second function returns
            - stack: memoryfunction1
    - redo-undo features (Photoshop, Editors)
- implementation:
    - using arrays
    - using linked lists
- the stack, used to save the variables for multiple functions, is called the call stack (e.g. recursion)

3.4 Queue

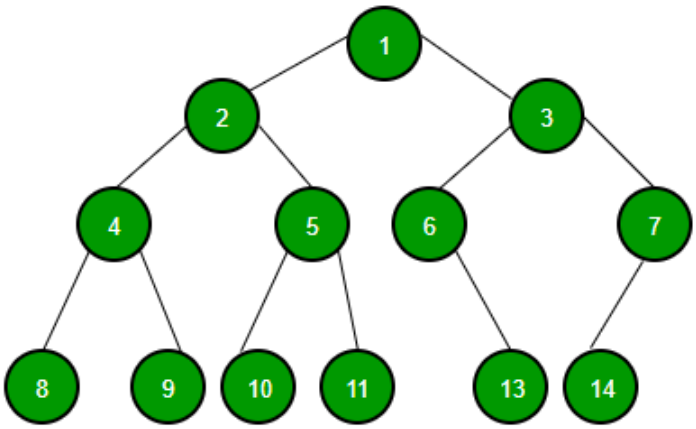
- linear data structure
- enqueue at the rear and dequeue at the front
- first in, first out
- enqueueing, dequeueing time: $O(1)$ (best case implementation)
- example
    - BFS queue
    - asynchronously transferred data
- implementation:
    - using arrays
    - using linked lists

3.5 Trees

- hierarchical data structure
- topmost node is the root
- elements with no children are called leaves
- trees do not have an upper limit on number of nodes as nodes are linked using pointers
- a tree whose elements have at most 2 children is called a binary tree
- example
    - folder structure of hard drives

Binary Search Trees

- node-based binary tree data structure
    - left subtree of a node contains only nodes with keys lesser than the node's key
    - right subtree of a node contains only nodes with keys greater than the node's key
    - left and right subtree each must be a bionary search tree
- each node contains:
    - data
    - pointer to left child
    - pointer to right child
- time complexity (worst case, h denotes the height of the tree):
    - searching $O(h)$
    - insertion $O(h)$
    - deletion $O(h)$
    - if the tree is perfectly balanced $h = logn$
    - if the tree has items that have been insorted in sorted order $h = n$



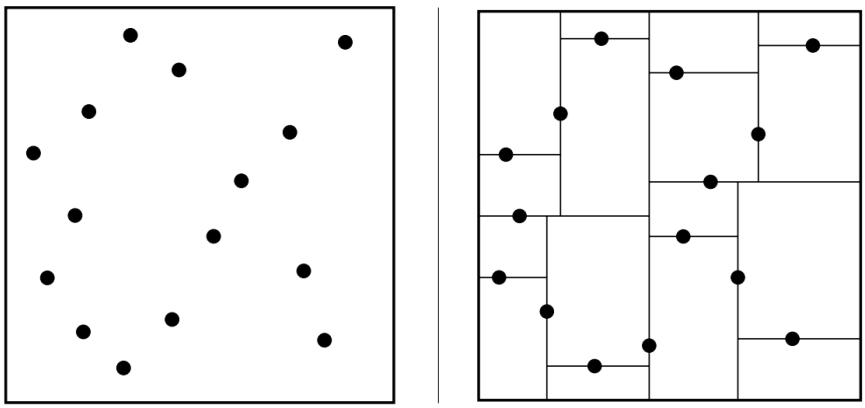
Source (https://www.geeksforgeeks.org/wp-content/uploads/binary-tree-to-DLL.png)

Self-Balancing Binary Search Trees

- automatically keeps the height small
  - adjust the tree after every insertion/deletion, so the maximum height is logarithmic ($log_2 n$)
- example implementations
  - red-black tree
  - splay tree

Kd-Trees

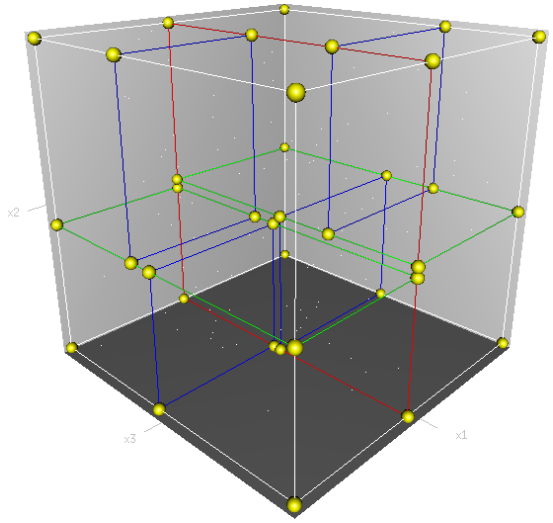
- a set S of n points or more complicated geometric objects in k dimensions
- solved problem: construct a tree that partitions space by half-planes such that each object is contained in its own box-shaped region



The Algorithm Design Manual, page 389

- this decomposing of space into smaller number of cells provides a fast way to access any object by position

- example for better understanding (assume dataset of 8 points):

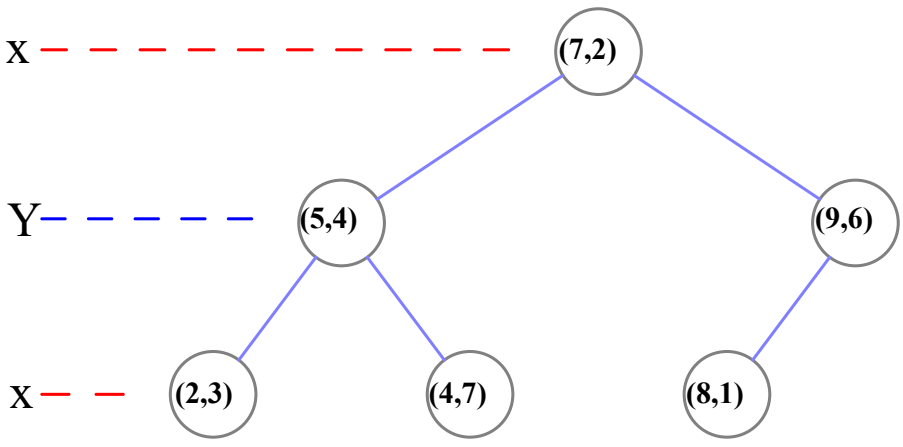


Source (https://upload.wikimedia.org/wikipedia/commons/b/b6/3dtree.png)

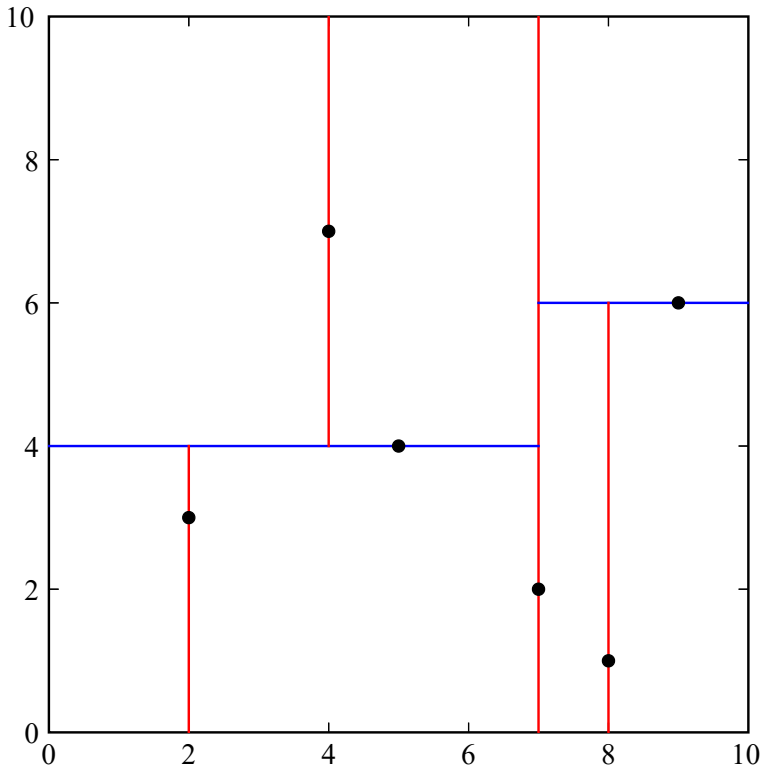
- 3-dimensional kd-tree
- first partition (red plane) cuts the root cell (whole cube) into two subcells
- green plane than splits the subcells of the red plane
- blue plane does the same for the subcells of the green plane
- we then get eight leaf cells
- each node is defined by a plane cutting through one of the dimensions
- nodes in subsets are partitioned again
- this process stops after lgn levels, with each point in its own leaf cell
  - if we have 8 points, we have partitioned all points after $lg8 = 3$ level
    - as we can see in the example above: 1. red plane 2. green plane 3. blue plane
- this example also shows that every box-shaped region is defined by $2k$ planes
  - here we have 6 planes for each leaf cell which is true because every leaf cell is a cube
  - if we would have a 2-dimension kd-tree we would have 4 planes for each leaf cell which also is true because a 2d cube is a rectangle and that is defined by 4 lines (one-dimensional planes)

another more concrete example:

- assume the data set (2,3), (5,4), (9,6), (4,7), (8,1), (7,2)
- to have a balanced tree it is efficient to select a root of the subtree that is (near) the median value of the axis selected
  - this can be done by using a sort algorithm to sort all points with respect to one axis and then get the median point of that sorted array
  - in this case: the point (7,2)
    - this means that the one-dimensional plane through the x-axis would cut the space in half
  - this will be recursively done with all subtrees until all points have been inserted in the kd-tree
  - the only difference is that the axis will change for every depth addition
    - this means that for the first node (depth 0) the x-axis will be the selected axis
    - for the second the y-axis
    - for the third the x-axis again because this is a 2-dimensional kd-tree
      - if this would be a 3-dimensional kd-tree it would be the z-axis
    - this can easily calculated by the formula `axis = depth % dimension_of_points`
      - depth refers to the tree depth
        - the root is at depth 0
        - children of the root are at depth 1
- the resulting kd-tree will look like this



Source (https://upload.wikimedia.org/wikipedia/commons/2/25/Tree_0001.svg)



Source (https://upload.wikimedia.org/wikipedia/commons/b/bf/Kdtree_2d.svg)

- the code implementation will be used in the "nearest neighbour search algorithm" paragraph

- kd-trees differ in exactly how the splitting plane is selected
- options include:
  - cycling through the dimensions
    - as shown in the example (x1 then x2 then x3 ...)
  - cutting along the largest dimension
  - quadtrees or octtrees
  - BSP-trees
  - R-trees
- high-dimensional spaces should be avoided
  - according to Steven S. Skiena, 2 up to ~20 are most useful

3.6 Priority Queue

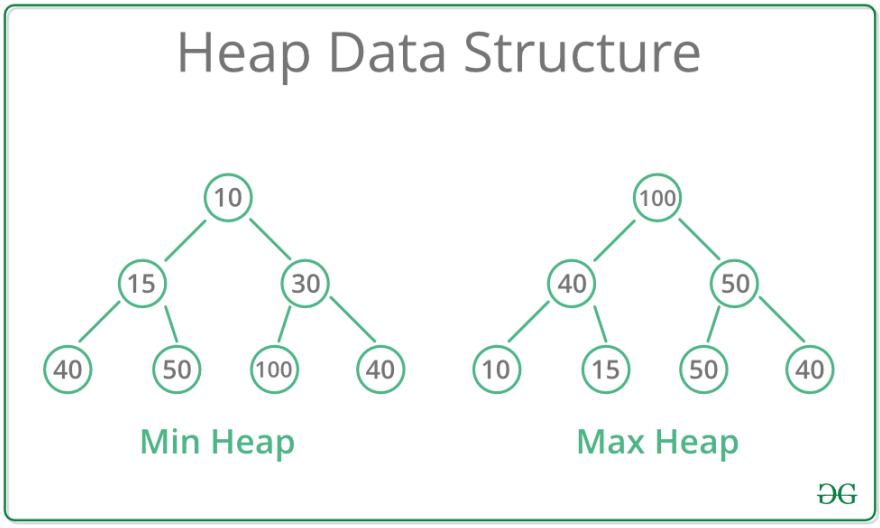
- set of records with numerically or otherwise totally-ordered keys
- solved problem: build and maintain a data structure for providing quick access to the smallest or largest key in the set
- each element of this data structure has a priority
- the basic priority queue supports three primary operations
  - insertion
  - find-maximum or minimum
  - delete-maximum or minimum
- example of a priority queue implemented with an
- unsorted array, sorted array, balanced binary search tree
  - the sorted array is in reverse order to have the minimum element as the tail element and to not move all elements when deleting or inserting the minimum

---	unsorted array	sorted array	balanced binary search tree
insert	O(1)	O(n)	O(logn)
find-minimum	O(1)	O(1)	O(1)
delete-minimum	O(n)	O(1)	O(logn)

- it is possible to achieve constant find-minimum time for each data structure by using an extra variable to store a pointer/index to the minimum entry
  - when asked to find it, we can simply return this value
- if delete-minimum happens, all entries have to be searched to find the new minimum and thus, require linear time on an unsorted array and logarithmic time on a tree

Heap

- data structure for efficiently supporting the priority queue operations insert and extract-min
- tree-based data structure in which the tree is a complete binary tree
- all levels of the tree, except possibly the last one, are fully filled and, if the last level of the tree is not complete, the nodes of that level are filled from left to right
- two types
  - max-heap
    - the key present at the root node must be greatest among the keys present at all of its children, this applies for all sub-trees
  - min-heap
    - the key present at the root node must be minimum among the keys present at all of its children, this applies for all sub-trees
- implementation
  - store each key in a node with pointers to its two children
    - memory heavy due to the pointers
  - store data as an array of keys
    - root of the tree at the first position
    - left and right children at the second and third position
    - this will store the 2^l keys of the lth level of the complete binary tree from left-to-right in positions 2^{l-1} to $2^l - 1$
    - these formulas assume a starting index of 1
    - the key at position k has its left child at position $2k$ and the right child at $2k + 1$
    - the height of the heap is $logn$
  - this second implementation does save memory but is less flexible than using pointers
  - the problem of this structure is that it cannot store arbitrary tree topologies without wasting large amounts of space
    - arbitrary tree topologies have sparse trees and this means that the array of keys would only work if empty entries are filled
  - the loss of flexibility explains why representing binary search trees is not a good idea but that it works just fine for heaps
- time complexity
  - insertion
    - worst case is $O(logn)$
  - deletion
    - worst case is $O(logn)$
  - search
    - worst case is $O(n)$
  - constructing a heap takes $O(nlogn)$ time



Source (https://www.geeksforgeeks.org/wp-content/uploads/MinHeapAndMaxHeap.png)

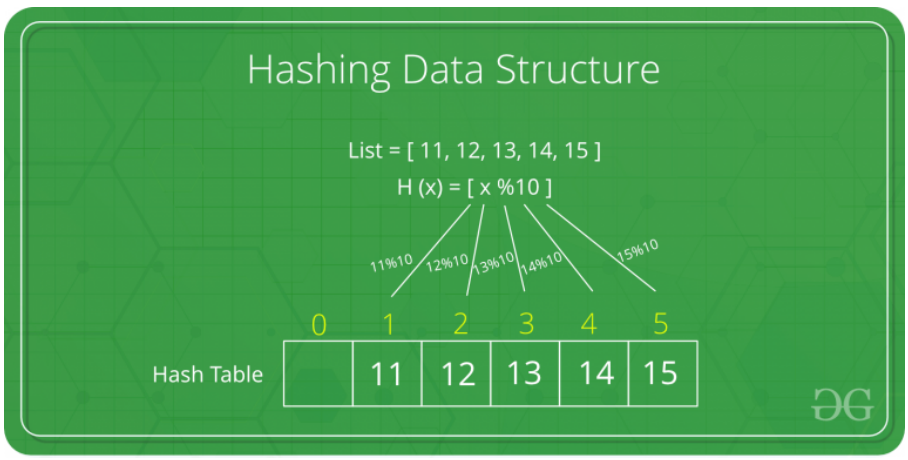
3.7 Dictionaries

- a set of records, each identified by one or more key fields
- abstract data type
  - implementation example: hash tables
- solved problem: build and maintain a data structure to efficiently locate, insert, and delete the record associated with any query key
- important questions for implementations
  - How many items?
  - What are the operation frequencies?
  - What is the access pattern for keys?
  - Should individual operations be fast or the total amount of work done over the entire program be minimized?

Chapter 12.1 in The Algorithm Design Manual

Hash Table

- uses a hash function
  - consistently maps a value with a particular key (hash values)
  - these hash values can be used to index a hash table
    - this process is called hashing
      - a key is converted into a small integer (hash value) using a hash function
        - can be considered to be constant $O(1)$
      - hash code is used to find an index (hashCode % arrSize) and the entire linked list at that index (chaining) is first searched for the presence of the key
        - worst-case time is $O(n)$
      - if found, its value is updated and if not, the key-value pair is stored as a new node in the linked-list
    - if there are n entries and the size of the array is b, the value n/b is called the load factor
      - needs to be kept low, so the number of entries at one index is less and the time complexity almost constant $O(1)$
- hash tables usually are arrays
- hash tables can also be a different data structure
  - e.g. balanced binary search trees
    - insertion, deletion, searching: $O(logn)$
    - Interesting explanation (https://stackoverflow.com/questions/22996474/why-implement-a-hashtable-with-a-binary-search-tree)
- example phonebook, caching (redis)
- search time: average case $O(1)$, worst case $O(n)$
- insert time: average case $O(1)$, worst case $O(n)$
- delete time: average case $O(1)$, worst case $O(n)$



Source (https://www.geeksforgeeks.org/wp-content/uploads/HashingDataStructure-min-768x384.png)

- collision (assigning a value to an array slot that has already been assigned to another value) solutions
  - chaining
    - make each cell of hash table point to a linked list of records that have same hash function value
    - devotes a considerable amount of memory to pointer
    - this can slow operations
  - open addressing
    - all elements are stored in the hash table itself (maintained as an array of elements)

        - each table entry contains either a record or NIL
        - when searching, table slots are examined one by one until the desired element is found or it is clear that the element is not in the table
        - linear probing:
            - linearly probe for the next slot
            - main problem is clustering of many consecutive elements
        - quadratic probing
            - taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found
        - double hashing
            - applying a second hash function to key when a collision occurs
    - avoiding collisions is taken care of by (good) hash functions
- rehasing
    - when the load factor increases to more than its pre-defined value (default is 0.75), the size of the array is increased (doubled)
    - all values are hashed again and stored in the new double sized array
- time complexity (n are items, m number of slots in the hash table)
    - chaining and open adressing require O(m) to initialize an m-element hash table to null elements prior to the first insertion
    - traversing chaining: O(n+m)
    - traversing open adressing: O(m)
    - chaining with doubly-linked lists:

	Hash table (expected)	Hash table (worst case)
Search(L, k)	$O(n/m)$	$O(n)$
Insert(L, x)	$O(1)$	$O(1)$
Delete(L, x)	$O(1)$	$O(1)$
Successor(L, x)	$O(n+m)$	$O(n+m)$
Predecessor(L, x)	$O(n+m)$	$O(n+m)$
Minimum(L)	$O(n+m)$	$O(n+m)$
Maximum(L)	$O(n+m)$	$O(n+m)$

The Algorithm Design Manual, page 90

3.8 Graphs

- non-linear data structure
- are made up of nodes (vertices) and edges (connecting lines)
- used to model how different things are connected to one another (network)
- solved problem: represent a graph using a flexible, efficient data structure

```
graph = {}
graph["Adam"] =["Philip", "Florencia"]
graph["Philip"] =[]
```

flavors of graphs:

- directed
    - the node "Adam" has one-way connections to the nodes "Philip" and "Florencia"
    - "Philip" has no one-way connections
    - example: maps (one-way streets)
- undirected
    - "Philip" would have a connection to "Adam"
- weighted
    - graphs have "costs" assigned to every edge
    - example: drive-time, length
- non-simple
    - contain self-loops
    - multiedge (edges occurs more than once in the graph)

- acyclic
  - contain no cycles
  - trees are connected, acyclic undirected graphs
- embedded
  - vertices and edges are assigned geometric positions (drawing of a graph)
  - any drawing of a graph is an embedding
    - may or may not have algorithmic significance
- topological
  - the structure of a graph is completely defined by the geometry of its embedding
- implicit
  - not constructed and then traversed
  - built as we use them
  - example: the state tree of the AI task where new states (nodes) were created by applying the state transition function
- explicit
  - explicitly constructed

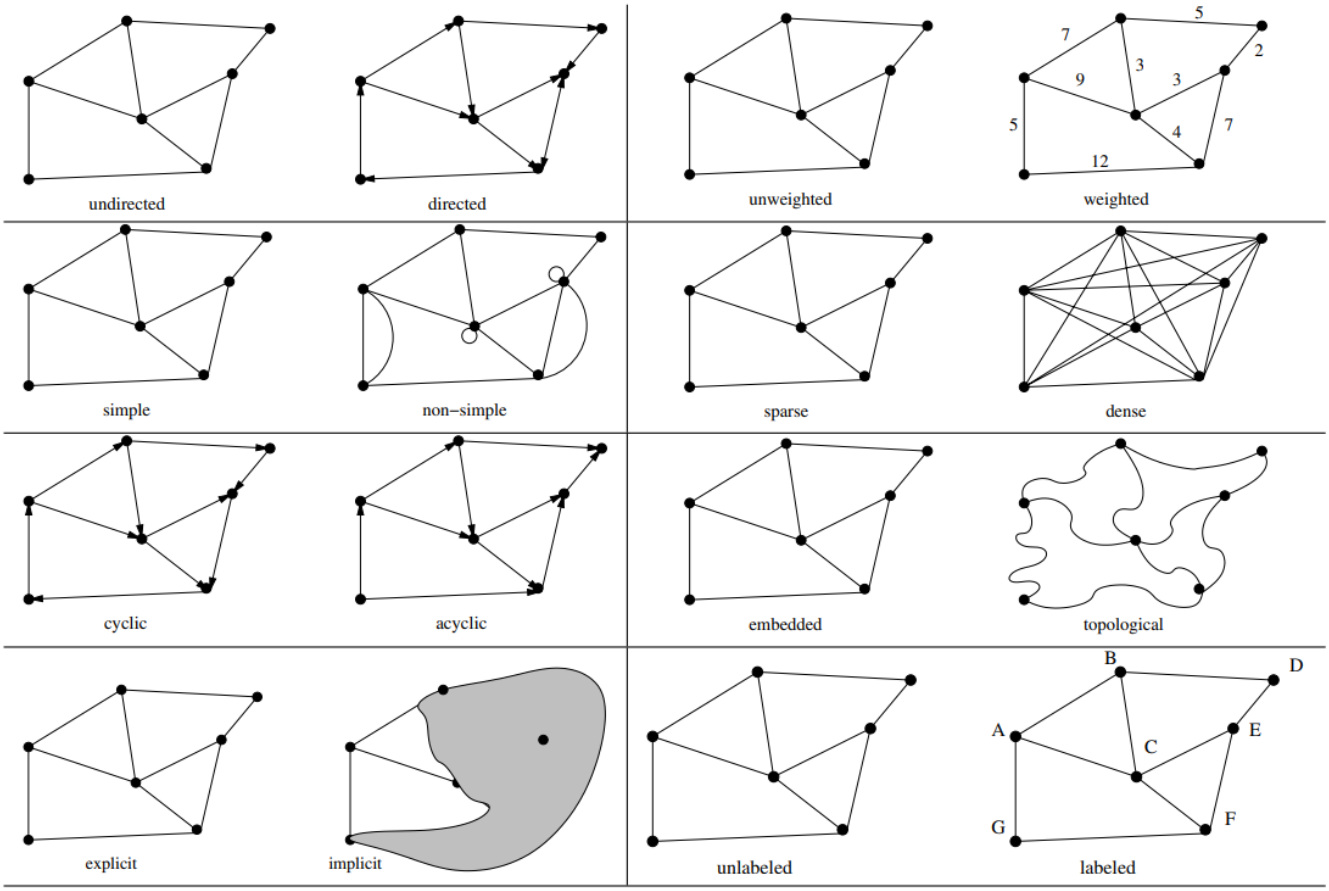
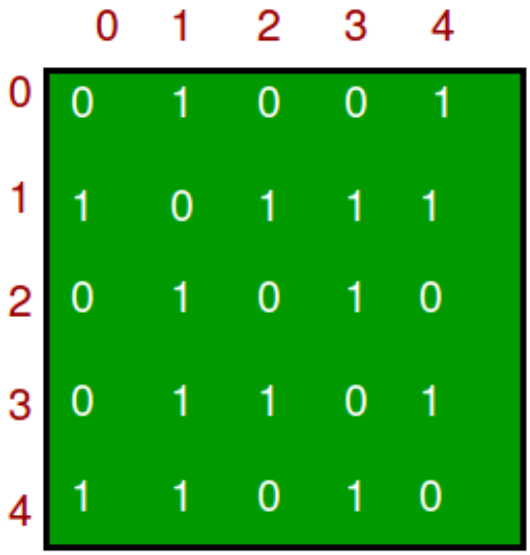


Figure 5.2: Important properties / flavors of graphs

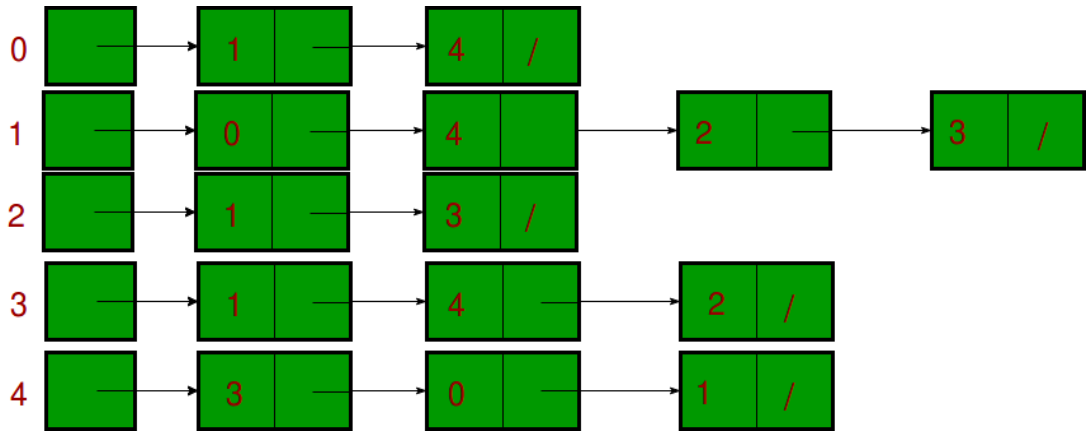
The Algorithm Design Manual, page 147

- two most commonly used representations of a graph
  - adjacency matrix
    - 2D array of size VxV, where V is the number of vertices
    - for an undirected graph always symmetric



Source (https://cdncontribute.geeksforgeeks.org/wp-content/uploads/adjacencymatrix.png)

- 1's represent that there is an edge between both vertices - 0's represent that there is no edge between both vertices - if this would be a weighted graph, the values (0 and 1) would be the weight value - Advantages: - removing an edge takes $O(1)$ time - edge testing takes $O(1)$ - Disadvantages: - consumes more space $O(V^2)$ - adding a vertex takes $O(V^2)$ - adjacency list - array of lists is used - size of the array is equal to V - an entry at array index i represents the list of vertices adjacent to the ith vertex - adjacency lists are the right data structure for most applications of graphs



Source (https://cdncontribute.geeksforgeeks.org/wp-content/uploads/listadjacency.png)

- questions to ask:
  - How big will your graph be?
    - adjacency matrices make sense for small graphs
  - How dense will your graph be?
    - adjacency matrices make sense for very dense graphs
  - Which algorithms will you be implementing?
  - Will you be modifying the graph over the course of your application?

Chapter 14.4 in The Algorithm Design Manual

A thing to add is this quote:

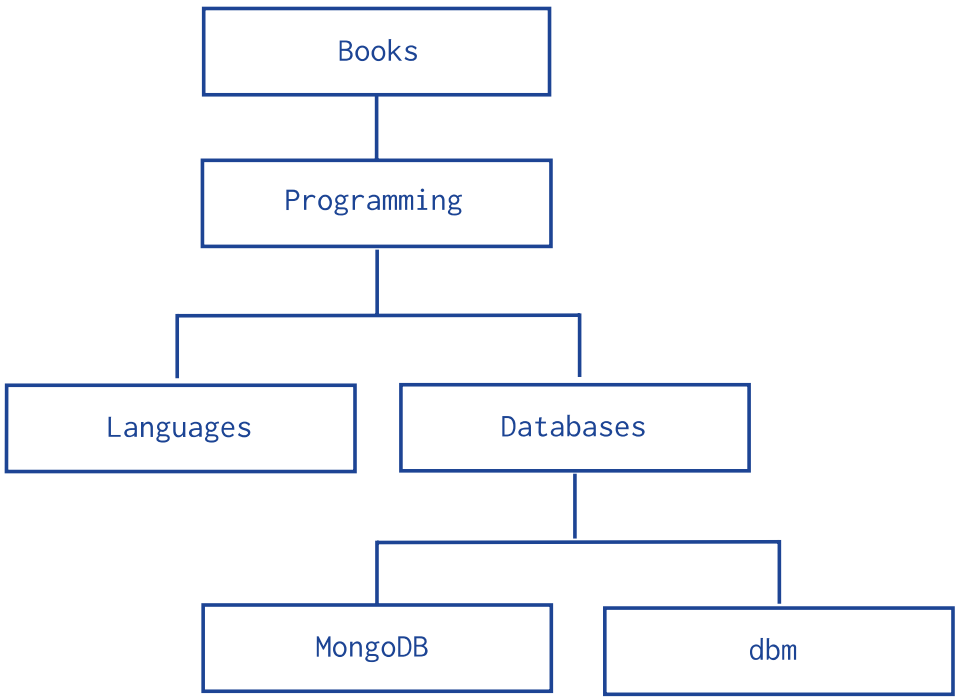
> Designing novel graph algorithms is very hard, so don't do it. Instead, try to design graphs that enable you to use classical algorithms to model your problem.
>
> - Skiena, Steven S. 1998. The Algorithm Design Manual (page 225)

4. Investigating a technology

Data is everywhere and data has to be saved. Databases solve this problem by providing a mechanism for storage and retrieval. NoSQL (“not only SQL”) databases have arisen since the early 21st century. The needs of Web 2.0 and the popularity of big data were calling for newer solutions than relational databases. I thought that investigating the technology might lead to interesting discovery in relation to complex data structures and their usage.
NoSQL databases are a huge topic and so I only focus on components where data structures and their applications are used. Furthermore, there are databases for specific use-cases. There is not one solution and some even say that it does not matter if one uses NoSQL or SQL as long as they know how to prepare their data.

The first kind of NoSQL database is called a document-oriented database. Documents have a JSON-like structure which, essentially, are combinations of key-value pairs. To relate this to our data structure part, we know that we have a data structure that can be used to save key-value pairs: A dictionary (implemented as a, e.g., hash table). Of course, this does not mean that any NoSQL database uses hash tables to save and retrieve data but this gives a nice addition to the theory of hash tables since the data structure can be used to achieve a similar result. A more complicated JSON structure can be achieved through nested key-value pairs. Moreover, if the value of a key-value pair is, for example, an array or other data structures, the document becomes very diverse and able to represent a lot of different data.
Another important NoSQL database type is the graph database. These databases use graph structures with nodes (vertices) and edges, as well as, properties to represent and store data. The underlying storage mechanism can be a key-value store or document-oriented approach but nevertheless, the graph data structure finds its application. In the example code of the graph section, the relationship is modeled by a hash table. This shows that a database can achieve a graph structure by using the hash table data structure.
To go deeper in the graph example, we can look at one of the biggest sectors in the world-wide-web: E-commerce. Consumption graphs are graphs that track the consumption of individual customers. This helps the companies to suggest products and predict future purchases. These graphs can be saved with the help of a graph database which shows again how important the data structures are.

Another important feature of a database is the ability to model large hierarchical or nested data relationships. To achieve that, databases use another data structure that was introduced in the previous chapter: Trees. There are different implementation methods to model this kind of data structure. One way is to reference the parent node in the child value. In a document-like value element, this could be a field with the key “parent” that holds the ID of the parent document. But the implementation does not matter for the application since the achieved structure is a tree that can be used by the database, users, and programmers to model data.



Source (https://docs.mongodb.com/manual/_images/data-model-tree.bakedsvg.svg)

I think that the example of databases shows the importance of data structures. What is also shows is that good knowledge of the possibilities of each is required to make rational decisions that help the program to work more efficiently, espcially if one plans to implement a database oneself. Furthermore, it also emphasizes the ability that different structures

can be achieved in different ways. For that, the knowledge about different data structures, their advantages, and disadvantages is necessary for a developer that handles data (which, in theory, is every program).

Another technology that we can investigate is an AI. In particular, the AI that plans to toast a bread by selecting the right action sequence. This was a task for the AI module and I showed the used algorithms in the algorihms chapter. Now I want to focus on the data structures that are used.

The first data structure that is used is a tuple. This is an immutable sequence that is typically used to store collections of heterogeneous data. There are also used if an immutable sequence of homogeneous data is needed. They can be compared with the array data structure even though they are mutable.

```
actions=(
    "plug_in_toaster",
    "unplug_toaster",
    "put_in_bread",
    "take_out_bread",
    "switch_on_toaster",
    "wait")
```

Python has a another data structure that comparable to the array data structure which is called a list. This is a mutable sequence of, typically, homogeneous data.
In the case of the AI application it does not matter if the data structure is a tuple or a list. The only thing that is done with the data structure is a looping through all of its items. Nevertheless, as tuples are immutable and the actions do not change, a tuple is the right choice. Furthermore, tuples are more efficient than lists because they are immutable (Source (https://stackoverflow.com/questions/68630/are-tuples-more-efficient-than-lists-in-python)). So it would be possible to use both, lists and tuples, but the right choice is the tuple data structure. This case shows, why it is important to know the data structures and when to use which.

The second data structure that is used is a Python dictionary.

```
state={
    "toaster_has_power":False,
    "toaster_is_on":False,
    "bread_location":"plate",
    "bread_state":"untoasted",
    "time":0
    }
```

This data structure is internally implemented as a hash table. Python dictionaries are unordered and mutable objects. In our case we are mapping a string to a value. Strings are hashable data types and thus, the dictionary can be used to map a value to the key string. We are also only accessing the values by their keys. This means that it does not matter that the dictionary is unordered. Thus, the dictionary data structure is the right choice. If the key would be a tuple, the hashing would also work due to their immutability. Lists would, therefore, not work.

The last data structure I want to highlight, it the open_list and close_list of the A* algorithm.

```
    open_list = []
    closed_list = []

    open_list.append(start_node)

    loop_counter = 0
    while(open_list):
        loop_counter += 1

        current_node = open_list[0]
        current_index = 0
        for index, item in enumerate(open_list):
            if item.f < current_node.f:
                current_node = item
                current_index = index

        # print(current_node.state)
        # print(current_node.f)

        open_list.pop(current_index)
        closed_list.append(current_node)
```

This code snippet clearly shows the reasoning for the chosen data structure. Both lists are Python list objects. As stated above, these are mutable sequences of, typically, homogeneous data. We can also see that we are appending nodes and popping nodes of the lists. This is not possible with a tuple. The Python set data structure could not have been used because they are unordered objects with no duplicates.

I think that this investigation shows nicely, why data structures are important and why it makes a lot of sense to know the theory about them. Furthermore, it emphasizes that the theory in a specific language is also important in order to reason about the right data structure implementation in that language.

---

5. Sources

- Skiena, Steven S. 1998. The Algorithm Design Manual: Text. Vol. 1. Springer Science & Business Media
- Bhargava, Aditya. 2016. Grokking Algorithms: An Illustrated Guide for Programmers and Other Curious People
- https://www.khanacademy.org/computing/computer-science/algorithms (https://www.khanacademy.org/computing/computer-science/algorithms)
- http://cslibrary.stanford.edu/103/LinkedListBasics.pdf (http://cslibrary.stanford.edu/103/LinkedListBasics.pdf)
- http://introcs.cs.princeton.edu/43stack/ (http://introcs.cs.princeton.edu/43stack/)
- https://stackoverflow.com/questions/7477181/array-based-vs-list-based-stacks-and-queues (https://stackoverflow.com/questions/7477181/array-based-vs-list-based-stacks-and-queues)
- http://www.cs.cornell.edu/courses/cs312/2004fa/lectures/lecture16.htm#:~:text=A%20good%20rule%20of%20thumb,*n (http://www.cs.cornell.edu/courses/cs312/2004fa/lectures/lecture16.htm#:~:text=A%20good%20rule%20of%20thumb,*
- https://www.geeksforgeeks.org/ (https://www.geeksforgeeks.org/)