Memory or Time

Jordan Fernando / 13510069
Informatics Engineering
School of Electrical Engineering and Informatics
Bandung Institute of Technology, Jl. Ganesha 10 Bandung 40132, Indonesia
jordan.fernando@students.itb.ac.id

Abstract**—**there are many ways to solve problems using algorithms. Most of the algorithms are implemented through computer by making programs. By implementing through computers, we need to use the limited memory and processing speed that are provided by the computer system itself. Some algorithms could be well balanced between the use of memory and the processing speed but there are also some algorithms that require much time or memory to process. Because of that, we need to choose one between memory usage and execution time.

Section 1 will talk about introduction explaining the title of this paper. Section 2 will talk about the basic theory needed to understand the content of this paper. Section 3 will talk about the facts of growth memory and processor performance. Section 4 will talk about the choice and the consideration for the choice. Section 5 will talk about some problems that can be time optimized. Section 6 will talk about the conclusion of this paper.

Index Terms**—**algorithms, limited, memory, time.

1. Introduction

In implementing algorithms through computer, we need to estimate the execution time and memory usage in running the algorithms as computer program since the memory and processing speed are limited. Some algorithms require much time or memory. In some of those, we need to choose between using a little memory or time in the processing. If we choose memory then it will cost a lot of time to process and if we choose time then it will cost a lot of memory to process. This kind of thing are the trade-offs that must be chosen especially in optimization.

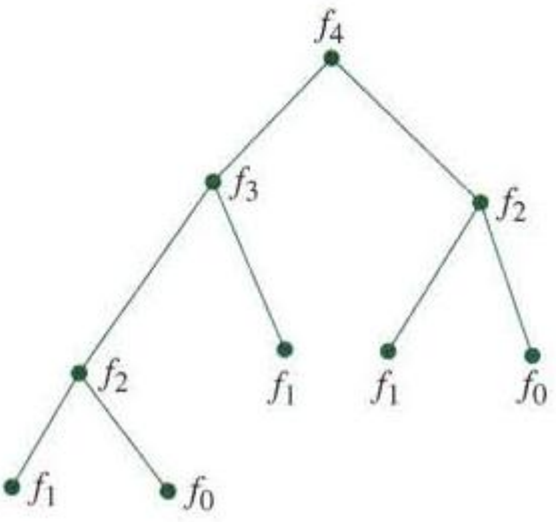
Optimization in program or software is the process of modifying a software system to make some aspect of it work more efficiently or use fewer resources. The optimization can be made at a number of levels such as design, source code, compile, assembly, and run time. Most of the optimization that programmers do is in the source code level because at source code level it is easy to control the memory usage and the execution time.

An example of optimization is *Fibonacci* sequence. In finding the Fibonacci sequence through algorithms we can just make it naïve by finding it recursively. Here is the naïve function to find the nth member.

```
function fib(n)
       if n = 0 return 0
       if n = 1 return 1
       return fib(n - 1) + fib(n - 2)
```

Example if we call fib(5), it will produce a sequence such as:

```
1. fib(4)
2. fib(3) + fib(2)
3. (fib(2) + fib(1)) + (fib(1) +
   fib(0))
4. (((fib(1) + fib(0)) + fib(1)) +
   (fib(1) + fib(0)))
```



We see that by just calling fib(4), the program will call fib(1) 3 times and fib(0) 2 times. By calling fib(n) it will lead to an exponential time algorithm. There is also other way of finding *Fibonacci* sequence such as:

```
   var m := map(0 → 0, 1 → 1)
   function fib(n)
       if map m does not contain key n
           m[n] := fib(n - 1) + fib(n
- 2)
       return m[n]
```

By using this algorithm we see that it only require O(n) execution time instead of exponential time but will consume O(n) space too. In a way to consume the same execution time but with less memory, there is also another way such as:

```
   function fib(n)
       var previousFib := 0,
currentFib := 1
       if n = 0
```

```
            return 0
        else repeat n - 1 times  //loop
is skipped if n=1
            var newFib := previousFib +
currentFib
            previousFib := currentFib
            currentFib  := newFib
        return currentFib
```

This one also require O(n) execution time and only consume O(1) space in memory.

Each of these two algorithms has their own advantages and disadvantages that we will discuss it later. This kind of optimization is also called *Dynamic Programming* which is a method for solving complex problems by breaking them down into simpler sub problems. We will discuss more about *Dynamic Programming* later (section 2.3).

In this paper, we will see which one to choose between memory usage and execution time of a program.

2. THEORY

2.1. *Complexity of Algorithm*

There are two kinds of complexity that has effect on complexity of algorithm. There are time complexity and space complexity.

2.1.1 Time Complexity

Time complexity can be expressed in terms of the number of operation used by the algorithm when the input has a particular size. The time complexity of algorithm also depends on the case of the input size. Based on the case, there are best-case scenario, average-case scenario, and worst-case scenario. From those case scenarios, people usually focus on the worst-case scenario because if the algorithm works good performance on worst-case scenario then it will also works good on average-case and even best-case scenario. We use the big O notation to determine the worst-case scenario.

A description of an algorithm in terms of big O notation usually provides the upper bound on the growth rate of the function. Beside the big O notation, there are several other notations in term of describing the complexity of an algorithm using the symbol such as o, Ω, ω, and Θ to describe other kinds of bounds. The complexity of an algorithm using big O notation usually classified as some types. Here is the table and chart that describes some types of big O notation.

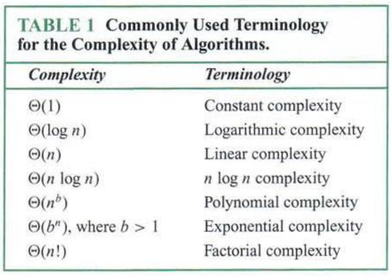
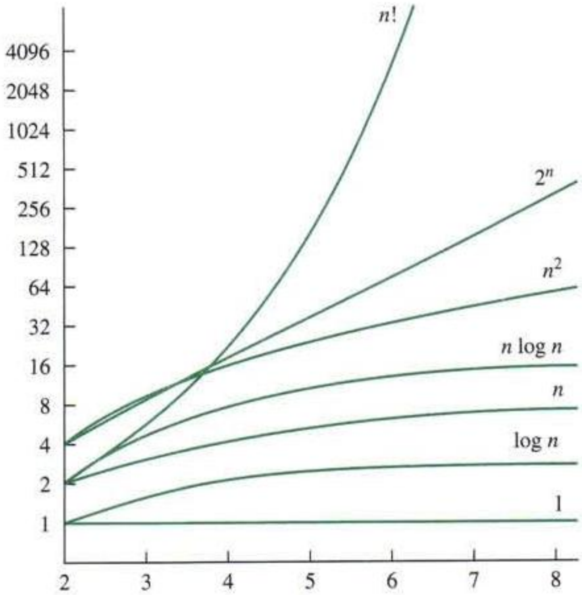


TABLE 1 Commonly Used Terminology for the Complexity of Algorithms.

Complexity	*Terminology*
$\Theta(1)$	Constant complexity
$\Theta(\log n)$	Logarithmic complexity
$\Theta(n)$	Linear complexity
$\Theta(n \log n)$	$n \log n$ complexity
$\Theta(n^b)$	Polynomial complexity
$\Theta(b^n)$, where $b > 1$	Exponential complexity
$\Theta(n!)$	Factorial complexity



As in the picture above, you can see that each complexity has a different growing rate of execution time as the input size get bigger. The O(n!) and O(2^n) are better than the others at small input (N<2) but become the worst at bigger input (N>2).

2.1.2 Space Complexity

Space complexity can be expressed in terms of the total memory used by the algorithm. Space complexity depends on the data structures used to implement the algorithm. There are many kinds of data structures such as array, vector, linked list, matrix, record, and many others.

2.2 *Recursive Algorithm*

Recursive algorithm is an algorithm that solves a problem by reducing it to instance of the same problem with the same input. The example of a recursive algorithm is as described before in section 1 for solving Fibonacci problem. Recursive algorithm consists of two parts which is basis and recurrence. Basis is a part of recursive algorithm that stops the recursive if the conditions are met. Recurrence is another part of recursive algorithm that calls again the function itself to solve the reduced problem. In order to proving recursive algorithms, we need to prove that the basis step is correct and the recurrence/inductive step is correct for reduced problem.

2.3 Dynamic Programming

As described in section 1, Dynamic Programming is a method for solving complex problems by breaking them down into simpler sub problems. Dynamic Programming itself can be achieved by using either of Top-down approach and Bottom-up approach.

2.3.1 Top-down approach

Top-down approach is an approach in dynamic programing that use recursive formulation to solve the problem by using the solution of its sub problems. However, some sub problems can be overlapping. In order to avoid finding the solution of the same sub problems, we can memorize and store the solution of the sub problems.

2.3.2 Bottom-up approach

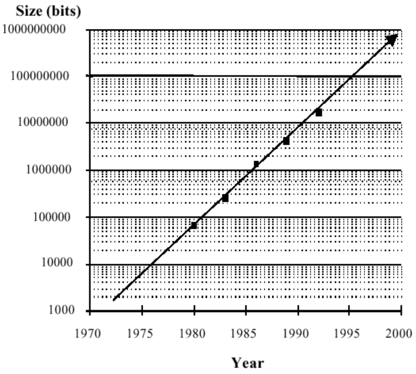
Bottom-up approach can be done once we already know the formulation of the recursive solution so that we can build the solution by using bottom up approach. First, we find the solutions to its sub problems and then use those solutions to build on and arrive at the solutions of the bigger sub problems.

3. Facts

Here are some facts about the growth of memory capacity and processor performance of a computer over time.

3.1 Growth of memory capacity

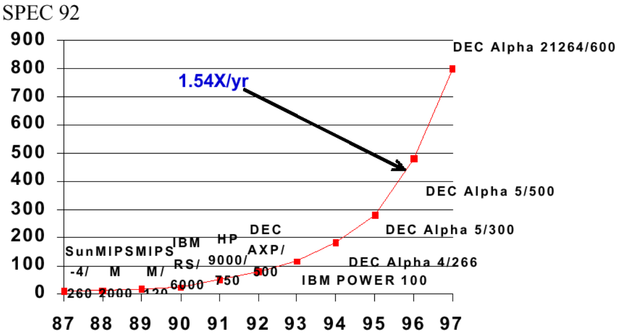
Memory capacity has approximately doubled every 1.5 years since the early 1970s. The growth of memory is exponential over time. Here is a chart that shows the growth of memory over time.



Year	Size(Mbit)
1980	0.0625
1983	0.25
1986	1
1989	4
1992	16
1996	64
2000	256

3.2 Growth of processor performance

The growth of processor performance is about 1.414 times per 1.5 years. The growth of processor performance is also exponential over time. Here is a chart that shows the growth of processor performance over times.



In the chart above, it can be seen that the increase is 1.54 times per year but that is only the raw increase. The real increase is 1.414 times every 1.5 years because the increase in processor performance also cost the performance to be used for some applications that cause the performance to increase itself.

4. The Choice And Consideration

From the data of facts (section 3), we can see that the facts support that the growth of memory capacity of computer is greater than the growth of processor performance over time. We also need to consider that the faster the processing speed, the hotter will the computer become and can cause damage to the computer part itself. If we consider it more, by using more memory for the algorithm we can make the algorithm works more efficient and consume less process using the method of *Dynamic Programming*. It is better if we choose time over memory because the efficiency is far better by using more memory and on nowadays program, the usage of memory is still minimum. Imagine if we use more memory for memorization, the process could become faster even more.

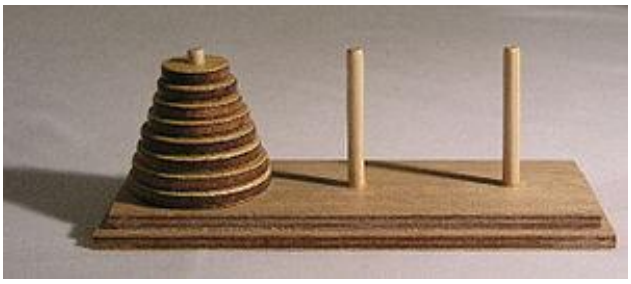
As in the example on section 1, we also can see there are two methods of bottom-up solution for Fibonacci sequence problem. The difference is that by using memory, we can use the memory for many cases that are asked for user. This kind of thing is called pre-computation. By using pre-computation method, the user may ask which Fibonacci sequence as much as the user wants. But it will be different that if we use the solution without memory. If the user asks for Fibonacci sequence many times then the program must redo the process all along for the input from the user.

Beside of that time is also an important part of our life because we can't rewind the time. Like example there is no way for us to just wait one day for finding Fibonacci sequence up to 1 billion each time we want to find Fibonacci number 1 billion. But the case will become different if we pre-compute it for one day and then we can use the memory over and over. The complexity for time also become O(1).

So we can say that optimization for time is better than for memory. Even though the trade-off is the consumption of memory become greater.

5. Some Problems For Time Optimization

5.1. Tower of Hanoi puzzle



The tower of Hanoi is a mathematical game or puzzle that consists of three rods, and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.

The objective of the puzzle is to move the entire stack to another rod, obeying the following rules:

- Only one disk may be moved at a time.
- Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
- No disk may be placed on top of a smaller disk.

We can see that the functional equation of this recursive solution is such as this.

$$\mathrm{S(n,h,t)\ =\ S(n-1,h,not(h,t))\ ;\ S(1,h,t)\ ;\ S(n-1,not(h,t),t)}$$

n denotes the number of disks to be moved, h denotes the home rod, t denotes the target rod, not(h,t) denotes the third rod (neither h nor t), “;” denotes concatenation.

S(n, h, t) is the solution to a problem consisting of n disks that are to be moved from rod h to rod t.

The basis is that if n=1 then we just move the disk from rod to target.

5.2. Factorial problem

The factorial problem is a problem that counts the factorial sequence given input of an integer number. We can see that the functional equation of this recursive solution is such as this.

$$f(n) = n * f(n-1)$$

n denotes the input of integer number given.

5.3. Shortest Path problem

Shortest path problem is a problem in finding the shortest path in a graph from a node to the target node. Normally we could just use recursive algorithms through all the nodes that have an adjacency to the initial node. But that would become too complex and go through all the combination of node available.

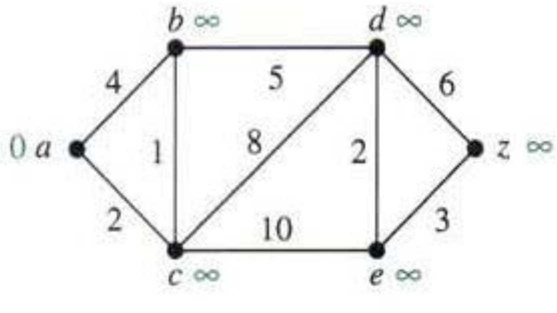
There is an algorithm in finding the shortest path problem in a given graph called the Djikstra algorithm. Djikstra algorithm also uses an optimization to memorize the distance between the source node and all other nodes. The algorithm for shortest path using Djikstra is like this.

```
 1  function Dijkstra(Graph, source):
 2      for each vertex v in Graph:           // Initializations
 3          dist[v] := infinity ;             // Unknown distance function from source to v
 4          previous[v] := undefined ;        // Previous node in optimal path from source
 5      end for ;
 6      dist[source] := 0 ;                   // Distance from source to source
 7      Q := the set of all nodes in Graph ;  // All nodes in the graph are unoptimized - thus are in Q
 8      while Q is not empty:                 // The main loop
 9          u := vertex in Q with smallest distance in dist[] ;
10          if dist[u] = infinity:
11              break ;                       // all remaining vertices are inaccessible from source
12          end if ;
13          remove u from Q ;
14          for each neighbor v of u:         // where v has not yet been removed from Q.
15              alt := dist[u] + dist_between(u, v) ;
16              if alt < dist[v]:             // Relax (u,v,a)
17                  dist[v] := alt ;
18                  previous[v] := u ;
19                  decrease-key v in Q;      // Reorder v in the Queue
20              end if ;
21          end for ;
22      end while ;
23      return dist[] ;
24  end Dijkstra.
```

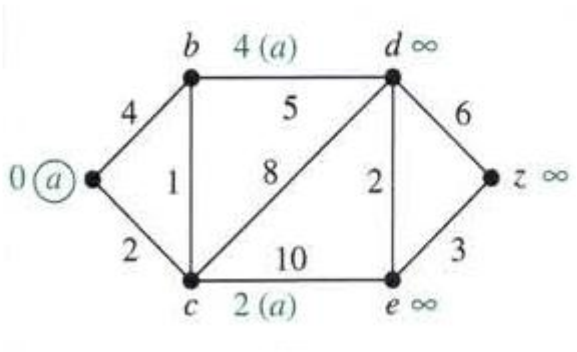
If we are only interested in a shortest path between vertices source and target, we can terminate the search at line 13 if u = target. Now we can read the shortest path from source to target by iteration:

```
S := empty sequence
u := target
while previous[u] is defined:
    insert u at the beginning of S
    u := previous[u]
end while ;
```

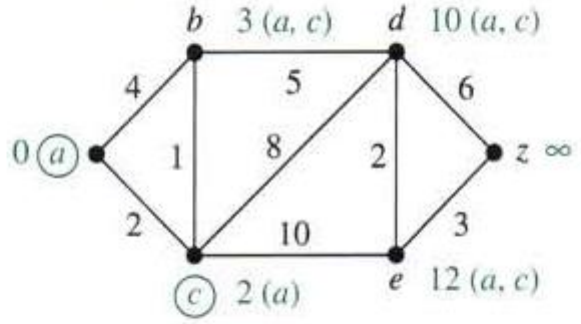
Here is the example sequence of finding the shortest path from node a to z given such graph.



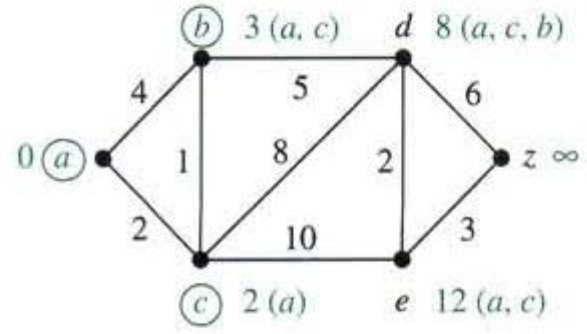
(a)



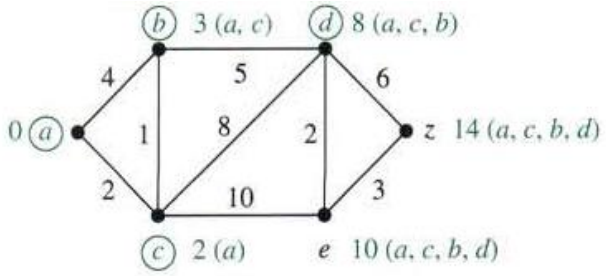
(b)



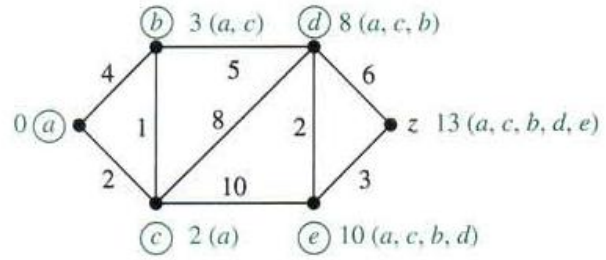
(c)



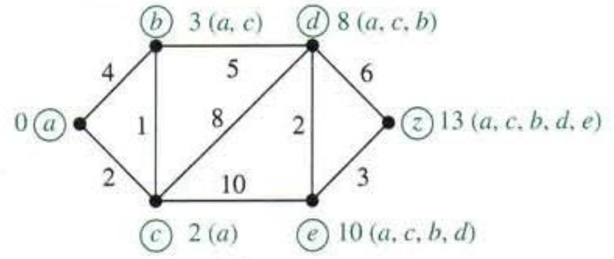
(d)



(e)



(f)



(g)

6. CONCLUSION

From the data-data on this paper, we can conclude tha optimization for time/process is better than optimization for memory/space.

REFERENCES

[1] Rosen, Kenneth H. “Discrete Mathematics and Its Applications Sixth Edition.” 2007. McGraw-Hill.
[2] http://en.wikipedia.org/wiki/Big_O_notation
[3] http://www.cise.ufl.edu/~mssz/CompOrg/CDAintro.html
[4] http://en.wikipedia.org/wiki/Dynamic_programming
[5] http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 11 Desember 2011

Jordan

Jordan Fernando (13510069)