BAB 2 LANDASAN TEORI

2.1. Algoritma

Algoritma secara sederhana merupakan urutan langkah- langkah logis untuk menyelesaikan masalah yang disusun secara sistematis.

2.1.1. Definisi Algoritma

Menurut Kamus Besar Bahasa Indonesia, algoritma adalah urutan logis pengambilan keputusan untuk pemecahan masalah.

2.1.2. Sifat-Sifat Algoritma

Donald E. Knuth menyatakan bahwa ada beberapa ciri-ciri algoritma [2], yaitu:

a. Algoritma mempunyai awal dan akhir . Suatu algoritma harus berhenti setelah mengerjakan serangkaian tugas atau dengan kata lain suatu algoritma harus memiliki langkah yang terbatas.
b. Setiap langkah harus didefinisikan dengan tepat sehingga tidak memiliki arti ganda.
c. Memiliki masukan atau kondisi awal.
d. Memiliki keluaran atau kondisi akhir.
e. Algoritma harus efektif; Jika diikuti dengan benar akan menyelesaikan permasalahan.

Berdasarkan cici-cirinya, algoritma dapat disimpulkan sifat utamanya yaitu:

a. *Input* : Suatu algoritma memiliki input atau kondisi awal sebelum algoritma dilaksanakan dan bisa berupa nilai-nilai peubah yang diambil dari himpunan khusus.
b. *Output* : Suatu algoritma akan menghasilkan output setelah dilaksanakan, atau algoritma akan mengubah kondisi awal menjadi kondisi akhir, dimana nilai output diperoleh dari nilai input yang telah diproses.
c. *Definiteness* : Langkah-langkah yang dituliskan dalam algoritma terdefinisi dengan jelas sehingga mudah dilaksanakan oleh pengguna algoritma.

d. *Finiteness* : Suatu algoritma harus memberi kondisi akhir atau output setelah melakukan sejumlah langkah yang terbatas jumlahnya untuk setiap kondisi awal atau input yang diberikan.
e. *Effectiveness* : Setiap langkah dalam algoritma dapat dilaksanakan dalam selang waktu tertentu sehingga pada akhirnya member solusi sesuai yang diharapkan.
f. *Generality* : langkah-langkah algoritma berlaku untuk setiap himpunan input yang sesuai dengan persoalan yang akan diberikan, tidak hanya untuk himpunan tertentu.

Algoritma sebagai langkah-langkah pemecahan masalah dapat dituliskan dengan berbagai cara, seperti:

1) Uraian deskriptif

   Penulisan algoritma dengan uraian deskriptif menggunakan bahasa yang biasa digunakan sehari-hari.

2) *Pseudocode*

   Algoritma yang dituliskan dalam kode-kode yang disepakati dan mempunyai urutan-urutan tertentu. Kode-kode ini dapat dikembangkan sendiri asalkan arti dari setiap kode disepakati bersama.

3) Bagan alir (*flowchart*)

   Flowchart (bagan alir dokumen) adalah penggambaran secara grafik dari langkah-langkah dan urutan-urutan prosedur dari suatu program.

2.2. Pencarian

Pencarian adalah proses pencarian solusi dari suatu permasalahan melalui sekumpulan kemungkinan ruang keadaan (*state space*). Ruang keadaan merupakan suatu ruang yang berisi semua keadaan yang mungkin. Kondisi suatu pencarian meliputi [3]:

1. Keadaan Awal.
2. Keadaan tujuan solusi yang telah dijangkau dan diperiksa apakah telah mencapai sasaran.
3. Biaya atau nilai yang diperoleh dari solusi.

Solusi merupakan suatu lintasan dari keadaan awal sampai keadaan tujuan. Secara umum, proses pencarian dapat dilakukan seperti berikut :

1. Memeriksa keadaan awal.
2. Mengeksekusi aksi yang telah ditetapkan untuk mencapai keadaan berikutnya.
3. Memeriksa jika keadaan baru merupakan solusinya. Jika ya, maka kondisi baru ditetapkan sebagai solusi. Jika tidak, keadaan baru tersebut menjadi keadaan sekarang. Proses nomor 2 dan 3 kembali dieksekusi dan diulang sampai solusi ditemukan atau ruang keadaan habis terpakai.

Berdasarkan ruang keadaan dari suatu permasalahan yang dicari, pencarian dibagi menjadi beberapa jenis, yaitu:

1. Pencarian *Uninformed*

   Sebuah algoritma pencarian uninformed adalah algoritma yang tidak mempertimbangkan sifat alami dari permasalahan [4]. Oleh karena itu algoritma tersebut dapat diimplementasikan secara umum, sehingga dengan implementasi yang sama dapat digunakan pada lingkup permasalahan yang luas, hal ini berkat abstraksi. Kekurangannya adalah sebagian besar ruang pencarian adalah sangat besar, dan sebuah pencarian *uninformed* (khususnya untuk pencarian pohon) membutuhkan banyak waktu walaupun hanya untuk contoh yang kecil. Sehingga untuk mempercepat proses, kadang-kadang hanya pencarian *informed* yang dapat melakukannya.

   a) Pencarian *List*

      Algoritma pencarian *list* adalah algoritma pencarian yang tujuannya adalah mencari sebuah elemen dari sebuah himpunan dengan suatu kunci (kemungkinan memuat informasi yang terkait dengan kunci). Oleh karena hal ini adalah masalah yang lazim dalam ilmu komputer, kompleksitas komputasi algoritma-algoritma tersebuh telah dipelajari dengan baik. Algoritma paling sederhana adalah pencarian *linear*, yang secara sederhana melihat setiap elemen dari list secara berurutan. Kompleksitas waktu pengerjaan algoritma ini adalah O(n), dimana n adalah banyaknya elemen dalam *list*, dan dapat digunakan langsung pada *list* yang belum diproses. Algoritma pencarian *list* yang lebih canggih adalah pencarian biner;

kompleksitas waktu pengerjaannya adalah O(log n). Waktu pengerjaannya jauh lebih baik daripada pencarian *linear* untuk *list* yang memiliki data banyak, tetapi sebelum dilakukan pencarian *list* terlebih dahulu harus terurut dan juga harus dapat diakses secara acak (pengaksesan acak). Pencarian interpolasi adalah lebih baik dari pencarian biner untuk *list* terurut yang sangat besar dan terdistribusi merata. Algoritma Grover adalah sebuah algoritma kuantum yang menawarkan percepatan kuadrat dibandingkan pencarian linear klasik untuk *list* tak terurut.

Tabel *hash* juga digunakan untuk pencarian *list*, hanya memerlukan waktu yang konstan untuk mencari pada kasus rata-rata, tetapi memiliki *overhead* ruang yang lebih dan pada kasus terburuk waktu pengerjaannya adalah O(n). Pencarian lain yang berdasarkan struktur data khusus, menggunakan pohon pencarian biner yang menyeimbangkan secara mandiri (*self-balancing binary search tree*) dan membutuhkan waktu pencarian O(log n); hal ini dapat dipandang sebagai pengembangan dari ide utama pencarian biner untuk memungkinkan penyisipan dan penghapusan yang cepat.

Sebagian besar algoritma pencarian, seperti pencarian *linear*, pencarian biner dan pohon pencarian biner yang bersifat *self-balancing*, dapat dikembangkan dengan sedikit tambahan untuk menemukan semua nilai yang kurang dari atau lebih dari sebuah kunci, operasi ini disebut pencarian jangkauan (*range search*). Pengecualin ada pada tabel *hash*, yang tidak dapat melakukan pencarian tersebut secara efisien.

b) Pencarian Pohon

Algoritma pencarian pohon adalah jantung dari teknik-teknik pencarian. Algoritma tersebut mencari *node* dari pohon, terlepas apakah pohon tersebut eksplisit atau implisit (dibangkitkan saat pengerjaan). Prinsip dasarnya adalah sebuah *node* diambil dari sebuah struktur data, suksesornya diperiksa dan ditambahkan pada struktur data. Dengan memanipulasi struktur data, pohon dieksplorasi dalam urutan yang berbeda-beda, diperiksa dari satu tingkat ke tingkat berikutnya (pencarian *Breadth-first*) atau mengunjungi node pucuk terlebih dahulu kemudian lacak balik/*backtracking* (pencarian

Depth-first). Contoh lain dari pencarian pohon antara lain pencarian *iterative-deepening*, pencarian berbatas kedalaman, pencarian dwiarah dan pencarian *uniform-cost.*

c) Pencarian Graf

Banyak permasalahan dalam teori graf dapat dipecahkan dengan memanfaatkan algoritma pencarian, seperti algoritma Dijkstra, algoritma Kruskal's, algoritma tetangga terdekat, dan algoritma Prim.-first, pencarian *iterative-deepening*, pencarian berbatas kedalaman, pencarian dwiarah dan pencarian *uniform-cost*.

2. Pencarian *Informed*

Pada pencarian *informed*, sebuah heuristik yang khusus untuk permasalahan tertentu digunakan sebagai pedoman. Sebuah heuristik yang baik dapat membuat sebuah pencarian *informed* bekerja secara dramatis melebihi pencarian *uninformed*.

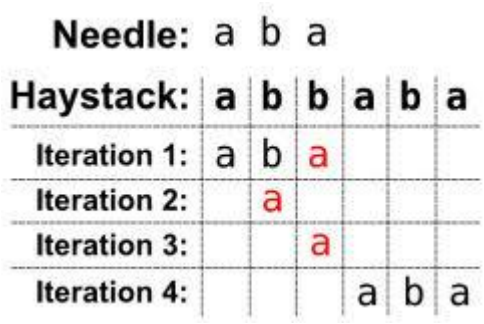
2.2.1. *String Matching* / Pencocokan Kata

String adalah susunan dari karakter (alphabet, angka, atau karakter lain) yang biasanya direpresentasikan sebagai struktur data *array*. Sedangkan *string matching* adalah sebuah permasalahan untuk menemukan pola susunan karakter *string* di dalam *string* lain atau bagian dari isi teks. Algoritma pencarian kata (*string*) adalah sebuah algoritma yang digunakan untuk mencari suatu pola dari suatu susunan huruf dalam sebuah kalimat [5].

2.2.1.1. Metode Pencarian *brute force*

Disebut juga metode *naive*. Sebuah metode pencarian sederhana namun tidak efisien yang digunakan untuk mencari letak posisi *string* yang dicari dalam sebuah *array string* yang lebih besar, satu per satu [6]. Pertama-tama kita melihat apakah ada salinan jarum dalam karakter pertama dari tumpukan jerami; jika tidak, kita melihat apakah ada salinan jarum mulai dari karakter kedua dari tumpukan jerami; jika tidak, kita terlihat mulai dari karakter ketiga, dan seterusnya. Dalam kasus normal, kita hanya harus melihat satu atau dua karakter

untuk setiap posisi yang salah untuk melihat bahwa itu adalah posisi yang salah, sehingga dalam kasus rata-rata, langkah-langkah ini memakan waktu O (n + m), dimana n adalah panjang tumpukan jerami dan m adalah panjang jarum; namun dalam kasus terburuk, mencari *string* seperti "aaaab" dalam *string* seperti "aaaaaaaaab", dibutuhkan O (nm).



Gambar 2.1. Contoh Pencarian naif / *brute force*

Berikut ini adalah beberapa kelebihan dan kekurangan dari algoritma *Brute Force*.

Kelebihan:

1) Metode *Brute Force* dapat digunakan untuk memecahkan hampir sebagian besar masalah (*wide application*).
2) Metode *Brute Force* sangat sederhana dan mudah dimengerti.
3) Metode *Brute Force* menghasilkan algoritma yang layak untuk beberapa masalah penting seperti pencarian, pengurutan, pencocokan *string*, perkalian matriks.
4) Metode *Brute Force* menghasilkan algoritma baku (*standard*) untuk tugas-tugas komputasi seperti penjumlahan / perkalian n buah bilangan, menentukan elemen minimum atau maksimum di dalam tabel (*list*).

Kekurangan:

1) Metode *Brute Force* jarang menghasilkan algoritma yang mangkus.
2) Beberapa implementasi algoritma *Brute Force* sangat lambat sehingga tidak dapat diterima.
3) Tidak konstruktif seperti metode lainnya.

Berikut ini adalah *pseudocode* dari algoritma *Brute Force*

```
procedure PencocokanString(
    input P : string,
    input T : string,
    n : integer,
   m :
)

Deklarasi:
i, j,next: integer
kmpNext : array[0..n] of interger

Algoritma:
preKMP(n, P, kmpNext)
i:=0
while (i<= m-n) do
    j:=0
    while (j < n and T[i+j] = P[j]) do
       j:=j+1
    endwhile
    if(j >= n) then
       ketemu[i]:=true;
    endif
    next:= j - kmpNext[j]
```

Gambar 2.2. *Pseudocode* metode *Brute Force*

2.2.1.2. Metode Pencarian *Knuth-Morris-Pratt*

Algoritma Knuth-Morris-Pratt (KMP) adalah salah satu algoritma pencarian *string*, dikembangkan secara terpisah oleh Donald E. Knuth pada tahun 1967 dan James H. Morris bersama Vaughan R. Pratt pada tahun 1966, namun keduanya mempublikasikannya secara bersamaan pada tahun 1977 [7].

Metode pencarian KMP bekerja dengan cara melewatkan iterasi-iterasi yang tidak perlu karena dinilai tidak akan menghasilkan kesesuaian antara pola/kata

yang dicari dengan susunan pola/kalimat utama. Misal sebuah pencarian berjumlah m di dalam sebuah kalimat K[] yang mengandung kata k[]. Algoritma yang paling mudah adalah dengan mencari kecocokan karakter pada nilai-nilai yang berurutan dari indeks m , posisi dalam *string* yang dicari , yaitu K[m]. Jika indeks m mencapai akhir dari *string* maka tidak ada karakter yang cocok, dalam hal pencarian dikatakan gagal. Pada setiap posisi m, algoritma mengecek keseusaian dari karakter pertama dalam kata yang dicari, yaitu K[m] = k[0]?. Jika kecocokan ditemukan, algoritma menguji karakter lain dalam mencari kata dengan memeriksa nilai-nilai yang berurutan dari posisi indeks kata, i. Algoritma mengambil karakter k[i] dalam mencari kata dan memeriksa kesetaraan ekspresi K[m+i] = k[i]?. Jika semua karakter yang berurutan sesuai dalam k pada posisi m maka kecocokan ditemukan pada posisi dalam *string* pencarian.

Dengan metode seperti itu, performa tidak dijamin optimal. Jika *string* tidak acak, kemudian memeriksa percobaan m dapat mengambil banyak perbandingan karakter. Kasus terburuk adalah jika dua *string* cocok dalam semua kecuali huruf terakhir. Bayangkan bahwa *string* K[] terdiri dari 1 milyar karakter yang semuanya A, dan bahwa kata k[] adalah 999 A karakter dan diakhiri dengan huruf B. Algoritma pencocokan *string* sederhana sekarang akan memeriksa 1000 karakter pada setiap posisi trial sebelum menolak hasil dan memajukan posisi trial pengecekan. Kini contoh sederhana pencarian akan mengambil sekitar 1000 perbandingan karakter kali 1 miliar untuk posisi 1 triliun perbandingan karakter. Jika panjang k [] adalah n, maka kinerja kasus terburuk adalah O (k · n).

Pada algoritma KMP, pergeseran untuk setiap pengecekan dapat dikurangi. Perhitungan penggeseran pada algoritma ini adalah sebagai berikut, bila terjadi ketidakcocokkan pada saat *pattern* sejajar dengan teks[i..i+n-1], kita bisa menganggap ketidakcocokan pertama terjadi di antara teks[i+j] dan *pattern*[j], dengan 0 < j < n. Berarti, teks[i..i+j-1] = *pattern*[0..j-1] dan a=teks[i+j] tidak sama dengan b=*pattern*[j]. Ketika kita menggeser, sangat beralasan bila ada sebuah awalan v dari *pattern* akan sama dengan sebagian akhiran u dari sebagian teks. Sehingga kita bisa menggeser *pattern* agar awalan v tersebut sejajar dengan akhiran dari u.

Dengan kata lain, pencocokkan *string* akan berjalan secara efisien bila kita mempunyai tabel yang menentukan berapa panjang kita seharusnya menggeser seandainya terdeteksi ketidakcocokkan di karakter ke-j dari *pattern*. Tabel itu harus memuat next[j] yang merupakan posisi karakter *pattern*[j] setelah digeser, sehingga kita bisa menggeser *pattern* sebesar j-next[j] relatif terhadap teks.

Secara sistematis, langkah-langkah yang dilakukan algoritma Knuth-Morris-Pratt pada saat mencocokkan *string*:

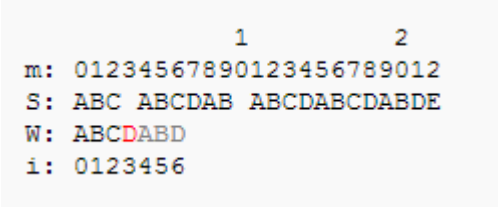
1. Algoritma Knuth-Morris-Pratt mulai mencocokkan *pattern* pada awal teks.
2. Dari kiri ke kanan, algoritma ini akan mencocokkan karakter per karakter *pattern* dengan karakter di teks yang bersesuaian, sampai salah satu kondisi berikut dipenuhi:

1. Karakter di *pattern* dan di teks yang dibandingkan tidak cocok (mismatch).
2. Semua karakter di *pattern* cocok. Kemudian algoritma akan memberitahukan penemuan di posisi ini.
3. Algoritma kemudian menggeser *pattern* berdasarkan tabel next, lalu mengulangi langkah 2 sampai *pattern* berada di ujung teks.

Untuk menggambarkan rincian algoritma, akan diberikan contoh kasus, dimana kata W = "ABCDABD" dan kalimat S = "ABC ABCDAB ABCDABCDABDE". Pada waktu tertentu, algoritma dalam keadaan yang ditentukan oleh dua variabel bilangan bulat:

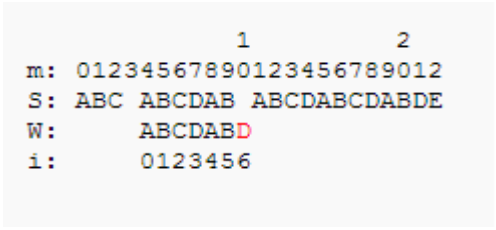
1. m yang menunjukkan posisi dalam S yang merupakan awal dari pertandingan prospektif untuk W
2. i indeks di W yang menunjukkan karakter saat ini sedang dipertimbangkan.

Dalam setiap langkah kita membandingkan S [m + i] dengan W [i] dan muka jika mereka sama. Hal ini digambarkan, pada awal menjalankan, seperti



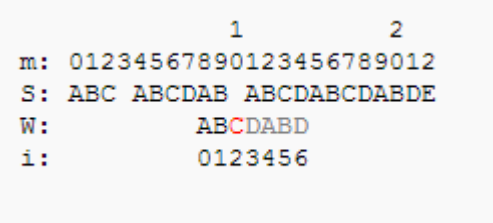
Gambar 2.3. Iterasi metode *Knuth-Morris-Pratt*

Kita lanjutkan dengan membandingkan karakter berturut-turut W untuk karakter dari S, bergerak dari satu ke yang berikutnya apakah mereka cocok. Namun, pada langkah keempat, kita mendapatkan S [3] adalah spasi dan W [3] = 'D', tidak cocok. Daripada mulai mencari lagi di S [1], sistem mencatat bahwa tidak ada 'A' terjadi antara posisi 0 dan 3 di S kecuali pada 0; oleh karena itu, setelah memeriksa semua karakter mereka yang sebelumnya, kita tahu tidak ada peluang untuk menemukan awal yang cocok jika kita memeriksa mereka lagi. Oleh karena itu kita beralih ke karakter berikutnya, menetapkan m = 4 dan i = 0.



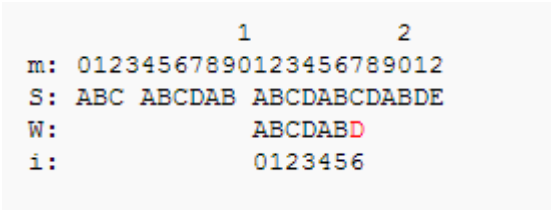
Gambar 2.4. Iterasi metode Knuth-Morris-Pratt #2

algoritma mendapatkan hasil yang hampir lengkap "ABCDAB" ketika, di W [6] (S [10]), sistem kembali memiliki ketidaksesuaian. Namun, sesaat sebelum berakhirnya pencocokan parsial saat ini, program melewati sebuah "AB" yang bisa menjadi awal dari pencocokan baru, jadi program harus mempertimbangkan ini. Seperti yang sudah kita tahu bahwa karakter ini cocok dengan dua karakter sebelum posisi saat ini, kita tidak perlu memeriksa mereka lagi; kita hanya me-reset m = 8, i = 2 dan terus cocok dengan karakter saat ini. Dengan demikian, kita tidak hanya menghilangkan karakter cocok sebelumnya dari S, tetapi juga cocok dengan karakter sebelumnya dari W.



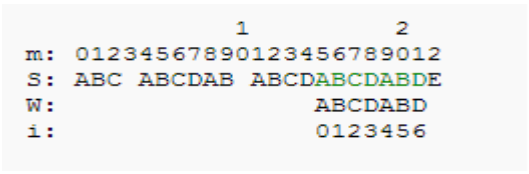
Gambar 2.5. Iterasi metode Knuth-Morris-Pratt #3

Pencarian ini gagal. namun, pola masih tidak mengandung spasi, sehingga dalam percobaan pertama, kita kembali ke awal W dan mulai mencari di karakter berikutnya S: m = 11, reset i = 0.



Gambar 2.6. Iterasi metode Knuth-Morris-Pratt #4

Kita kembali mendapatkan hasil "ABCDAB" tapi karakter berikutnya, 'C', tidak cocok dengan karakter akhir 'D' dari kata W. Penalaran seperti sebelumnya, kita menetapkan m = 15, untuk memulai pada dua-karakter *string* "AB" yang mengarah ke posisi saat ini, set i = 2, dan terus mencocokkan dari posisi saat ini.



Gambar 2.7. Iterasi metode Knuth-Morris-Pratt #5

Kali ini kita mampu menyelesaikan seluruh pencocokkan, yang karakter pertama adalah S[15].

Berikut adalah *pseudocode* algoritma KMP pada fase pra-pencarian:

```
procedure preKMP(
    input P : array[0..n-1] of char,
    input n : integer,
    input/output kmpNext : array[0..n] of integer
)
Deklarasi:
i,j: integer

Algoritma
  i := 0;
  j := kmpNext[0] := -1;
  while (i < n) {
     while (j > -1 and not(P[i] = P[j]))
        j := kmpNext[j];
     i:= i+1;
     j:= j+1;
     if (P[i] = P[j])
        kmpNext[i] := kmpNext[j];
     else
        kmpNext[i] := j;
     endif
  endwhile
```

Gambar 2.8. *Pseudocode* algoritma KMP pada fase pra-pencarian

Dan berikut adalah *pseudocode* algoritma KMP pada fase pencarian:

```
procedure KMPSearch(
    input m, n : integer,
    input P : array[0..n-1] of char,
    input T : array[0..m-1] of char,
    output ketemu : array[0..m-1] of boolean
)

Deklarasi:
i, j,next: integer
kmpNext : array[0..n] of interger

Algoritma:
preKMP(n, P, kmpNext)
i:=0
while (i<= m-n) do
    j:=0
    while (j < n and T[i+j] = P[j]) do
       j:=j+1
    endwhile
    if(j >= n) then
       ketemu[i]:=true;
    endif
    next:= j - kmpNext[j]
    i:= i+next
endwhile
```

Gambar 2.9. *Pseudocode* algoritma KMP pada fase pencarian

Algoritma ini menemukan semua kemunculan dari *pattern* dengan panjang n di dalam teks dengan panjang m dengan kompleksitas waktu O(m+n). Algoritma ini hanya membutuhkan O(n) ruang dari memori *internal* jika teks dibaca dari *file eksternal*. Semua besaran O tersebut tidak tergantung pada besarnya ruang alfabet.

2.3. Pengujian Perangkat Lunak

Pengujian perangkat lunak di bidang Rekayasa Perangkat Lunak adalah proses dalam siklus hidup sebuah perangkat lunak yang memastikan bahwa produk perangkat lunak memenuhi kualitas yang diharapkan dan memastikan bahwa perangkat lunak memenuhi persyaratan spesifikasi. Pengujian perangkat lunak dimaksudkan untuk menemukan cacat (*bug*) dalam sebuah program, meskipun metode pengujian yang diberikan tidak dapat menjamin untuk menemukan semua cacat. Dengan demikian, umumnya bagi aplikasi dilakukan berbagai metodologi pengujian selama siklus hidup perangkat lunak, seperti unit testing selama pengembangan, pengujian integrasi setelah modul dan sistem selesai, dan pengujian penerimaan pengguna untuk memungkinkan para pemangku kepentingan untuk menentukan apakah kebutuhan mereka telah terpenuhi.

Unit pengujian adalah jenis pengujian perangkat lunak yang melibatkan persiapan tes yang terdefinisi dengan baik untuk fungsi prosedural dari sebuah program yang memberikan kepastian bahwa modul atau fungsi telah berperilaku sebagaimana dimaksud. Tes unit yang disebut sebagai tes '*white-box*' (kebalikan dari tes ‘*black-box*') karena mereka ditulis dengan pengetahuan penuh dari struktur internal fungsi dan modul di bawah tes. Tes unit biasanya disiapkan oleh pengembang yang menulis kode yang diuji dan biasanya berjalan otomatis yang dijalankan oleh kerangka unit testing (seperti JUnit untuk Java atau kerangka Uji di Ruby). Tujuannya bukan untuk menguji setiap jalur eksekusi dalam unit, melainkan untuk fokus pada bidang risiko, ketidakpastian, atau kekritisan. Setiap tes berfokus pada satu aspek dari kode (uji satu hal) dan biasanya diatur dalam *suite tes* kesamaan.

Beberapa manfaat dari *unit testing* meliputi:

1. Dokumentasi: Penyusunan suite tes untuk sistem tertentu menyediakan jenis dokumentasi pemrograman menyoroti perilaku yang diharapkan dari fungsi dan modul dan memberikan contoh bagaimana berinteraksi dengan komponen kunci.

2. Keterbacaan: Unit pengujian mendorong gaya pemrograman modul kecil, masukan yang jelas dan output dan lebih sedikit ketergantungan antar komponen. Kode yang ditulis untuk memudahkan pengujian (testability) mungkin lebih mudah untuk membaca dan mengikuti.
3. Regresi: Bersama-sama, *suite tes* dapat dilaksanakan sebagai uji regresi dari sistem. Otomatisasi tes berarti bahwa setiap cacat yang disebabkan oleh perubahan pada kode dengan mudah dapat diidentifikasi. Ketika cacat ditemukan bahwa menyelinap melalui, tes baru dapat ditulis untuk memastikan itu akan diidentifikasi di masa depan.

Unit test secara tradisional ditulis setelah program selesai. Sebuah alternatif yang populer dalah untuk mempersiapkan tes sebelum fungsionalitas dari aplikasi disiapkan, disebut Test-Pertama atau Test-Driven Development (TDD). Dalam metode ini, tes tertulis dan diaksanakan, gagal sampai fungsionalitas aplikasi ditulis untuk membuat tes lulus. Persiapan awaltes memungkinkan programmer untuk mempertimbangkan perilaku yang diperlukan dari program dan antarmuka dan fungsi program perlu mengekspos sebelum mereka ditulis.

Kekhawatiran pengujian perangkat lunak yang sangat relevan dengan perkembangan, investigasi, dan penerapan metaheuristik dan Kecerdasan Komputasional algoritma. Secara khusus, budaya yang kuat dari investigasi empiris dan pembangunan berbasis prototipe menuntut tingkat dasar kepercayaan dalam sistem yang disajikan dalam artikel dan makalah. Dipercaya dapat ditanamkan dalam suatu algoritma dengan menilai kualitas pelaksanaan algoritma itu sendiri. Unit pengujian ringan (hanya membutuhkan penulisan kode tes otomatis) dan memenuhi kebutuhan mempromosikan kualitas dan kepercayaan dalam kode sementara prototipe dan mengembangkan algoritma. Sangat disarankan sebagai langkah dalam proses penelitian algoritma empiris di bidang metaheuristik, Kecerdasan Komputasional, dan biologis Terinspirasi Komputasi.

2.3.1. *Rules-of-Thumb*

Pengujian unit cukup mudah, meskipun menulis *unit test* yang baik itu sulit dikarenakan hubungan yang kompleks pada tiap tes dengan kode yang diuji. Pengujian metaheuristik dan *Computational Intelligence algorithms* lebih sulit lagi mengingat sifat probabilistik mereka dan kemampuan mereka untuk '*work in spite of you*', yaitu, memberikan semacam hasil bahkan ketika diimplementasikan dengan adanya *bug*.

Pedoman berikut dapat membantu ketika unit pengujian algoritma:

1) *Start Small*: Beberapa *unit test* lebih baik daripada tidak ada sama sekali dan setiap tes tambahan dapat meningkatkan kepercayaan dan kualitas *script*. Untuk implementasi algoritma yang ada, mulai dengan menulis tes untuk perilaku kecil dan sederhana dan perlahan-lahan membangun sebuah *test suite.*
2) *Test One Thing*: Setiap pengujian harus fokus pada verifikasi perilaku salah satu aspek dari satu unit kode. Menulis ringkas dan perilaku yang berfokus unit test adalah tujuan metodologi.
3) *Test Once*: Sebuah perilaku atau harapan hanya perlu diuji sekali, jangan mengulang ujian setiap kali suatu unit tertentu diuji.
4) *Don't forget the I/O*: Ingatlah untuk menguji masukan dan keluaran dari unit kode, khususnya pra-kondisi dan pasca-kondisi. Hal ini dapat mudah untuk fokus pada poin keputusan dalam unit dan melupakan tujuan utamanya.
5) *Write code for testability*: Pengujian harus membantu membentuk kode yang mereka uji. Tulis fungsi kecil atau modul, berpikir tentang pengujian saat menulis kode (atau menulis tes pertama), dan kode refactor (kode update setelah fakta) untuk membuatnya lebih mudah untuk menguji.
6) *Function independence*: Mencoba untuk membatasi ketergantungan langsung antara fungsi, modul, objek dan konstruksi lainnya. Hal ini terkait dengan testability dan menulis fungsi kecil meskipun menunjukkan batas berapa banyak interaksi yang ada antara unit kode dalam algoritma. Kurang ketergantungan berarti kurang efek samping dari suatu unit tertentu dari kode dan tes akhirnya kurang rumit.

7) *Test Independence*: Uji harus independen dari satu sama lain. Kerangka memberikan kait untuk set-up dan air mata-down negara sebelum pelaksanaan setiap tes, seharusnya tidak ada perlu memiliki satu tes menyiapkan data atau negara untuk tes lainnya. Tes harus dapat melaksanakan secara mandiri dan dalam urutan apapun.
8) *Test your own code*: Hindari menulis tes yang memastikan perilaku *framework* atau *library*, seperti keacakan nomor acak generator atau apakah matematika atau fungsi *string* berperilaku seperti yang diharapkan. Fokus pada menulis tes untuk manipulasi data yang dilakukan oleh kode yang Anda tulis.
9) *Probabilistic testing*: Metaheuristik dan Komputasi Intelijen algoritma umumnya menggunakan stokastik atau keputusan probabilistik. Ini berarti bahwa beberapa perilaku yang tidak deterministik dan lebih sulit untuk menguji. Seperti contoh, menulis tes probabilistik untuk memverifikasi bahwa proses seperti berperilaku sebagaimana dimaksud. Mengingat bahwa tes probabilistik lebih lemah daripada tes deterministik, pertimbangkan untuk menulis tes deterministik pertama. Sebuah perilaku probabilistik dapat dibuat deterministik dengan mengganti nomor acak dengan *proxy* yang mengembalikan nilai-nilai deterministik, disebut mock. Tingkat pengujian mungkin memerlukan dampak lebih lanjut untuk kode asli untuk memungkinkan modul yang bersangkutan dan objek-objek yang diteliti.
10) *Consider test-first*: Menulis tes pertama kali dapat membantu mengerucutkan harapan ketika menerapkan algoritma dari literatur, dan membantu untuk memperkuat pikiran ketika mengembangkan atau prototipe ide baru.

2.4. Notasi *Big-O*

Notasi O besar (*big-O notation*) atau notasi Landau adalah notasi matematika yang digunakan terutama pada bidang ilmu komputer dan matematika. Notasi ini digunakan untuk menyatakan keefektifan sebuah algoritma [8]. Notasi ini bekerja dengan cara memperhitungkan input yang diberikan oleh user. *Big-O* digunakan dalam bidang Ilmu Komputer untuk menggambarkan

kinerja atau kompleksitas dari sebuah algoritma. *Big-O* menggambarkan skenario terburuk dari sebuah fungsi, dan dapat digunakan untuk menggambarkan waktu eksekusi yang dibutuhkan atau ruang yang digunakan (misalnya dalam memori atau pada disk) oleh algoritma.

Beberpa notasi standar *Big-O* dalam teori kompleksitas waktu pada algoritma pencarian adalah sebagai berikut:

1. O(1)

   Notasi O(1) menandakan sebuah algoritma yang mempunyai kompleksitas waktu yang konstan terlepas dari berapa besar masukan yang diproses.

2. O(n)

   O (N) menggambarkan suatu algoritma yang kinerjanya akan tumbuh secara linear dan dalam proporsi langsung dengan ukuran set masukan data.

3. O(n^2)

   O (N2) merupakan algoritma yang kinerjanya berbanding lurus dengan kuadrat ukuran set masukan data. Hal ini biasa terjadi dengan algoritma yang melibatkan iterasi bersarang di atas kumpulan data.

4. O(log N)

   Notasi ini akan dicontohkan dengan algoritma pencarian biner. Pencarian biner adalah teknik yang digunakan untuk mencari set data berurut. Ia bekerja dengan memilih elemen tengah dari kumpulan data, pada dasarnya median, dan membandingkannya terhadap nilai target. Jika nilai-nilainya cocok maka pencarian dianggap sukses. Jika nilai target lebih tinggi dari nilai elemen penyelidikan itu akan mengambil bagian atas kumpulan data dan melakukan operasi yang sama terhadap hal itu. Demikian juga, jika nilai target lebih rendah dari nilai elemen penyelidikan itu akan melakukan operasi terhadap bagian bawah. Ini akan terus membagi data set dengan setiap iterasi sampai nilai telah ditemukan atau sampai tidak lagi dapat membagi kumpulan data.

   Jenis algoritma digambarkan sebagai O (log N). Mengurangi separuh berulang set data yang dijelaskan dalam contoh pencarian biner menghasilkan kurva pertumbuhan yang puncak di awal dan perlahan-lahan mendatar sebagai ukuran data set meningkatkan misalnya set input data yang berisi 10 item

membutuhkan waktu satu detik untuk menyelesaikan, satu set data yang berisi 100 item memakan waktu dua detik, dan satu set data yang berisi 1000 item akan memakan waktu tiga detik. Menggandakan ukuran kumpulan data input memiliki sedikit efek pada pertumbuhan sebagai setelah iterasi tunggal dari algoritma kumpulan data akan dibagi dua dan karena itu setara dengan input data set setengah ukuran. Hal ini membuat algoritma seperti pencarian biner yang sangat efisien ketika berhadapan dengan set data yang besar.

2.5. Pengertian *Game*

Game berasal dari kata bahasa inggris yang memiliki arti dasar Permainan. Permainan dalam hal ini merujuk pada pengertian "kelincahan intelektual" (intellectual playability). Game juga bisa diartikan sebagai arena keputusan dan aksi pemainnya. Ada target-target yang ingin dicapai pemainnya. Kelincahan intelektual, pada tingkat tertentu, merupakan ukuran sejauh mana game itu menarik untuk dimainkan secara maksimal.

Pada awalnya, game identik dengan permainan anak-anak. Kita selalu berpikir game merupakan suatu kegiatan yang dilakukan oleh anak-anak yang dapat menyenangkan hati mereka.

Dengan kata lain, segala bentuk kegiatan yang memerlukan pemikiran, kelincahan intelektual dan pencapaian terhadap target tertentu dapat dikatakan sebagai game.

Tetapi yang akan dibahas pada kesempatan ini adalah game yang terdapat di komputer, baik off line maupun online. Saat ini perkembangan games di komputer sangat cepat. Para pengelola industri game berlomba-lomba untuk menciptakan game yang lebih nyata dan menarik untuk para pemainnya. Hal inilah yang membuat perkembangan games di komputer sangat cepat. Sehingga games bukan hanya sekedar permainan untuk mengisi waktu luang atau sekedar hobi. Melainkan sebuah cara untuk meningkatkan kreatifitas dan tingkat intelektual para penggunanya.

Jadi, bermain game adalah suatu proses "fine tuning" (atau penyamaan frekuensi) dari logika berpikir anak-anak kita dengan logika berpikir aplikasi

komputer yang canggih tadi. Pada saat bersamaan, game juga secara nyata mempertajam daya analisis para penggunanya untuk mengolah informasi dan mengambil keputusan cepat yang jitu. Namun, tentu saja kenyataan juga harus kita masukkan kedalam perhitungan. Kenyataan itu diantaranya adalah kecanduan para pemain / penggunanya yang akut terhadap permainan komputer semacam ini. Mereka bisa lupa segala-galanya akan tugas mereka yang lain termasuk tugas menuntut ilmu**.**

Menurut Agustinus Nilwan dalam bukunya "Pemrograman Animasi dan Game Profesional" terbitan Elex Media Komputindo, game di artikan sebagai suatu aktivitas tersetruktur atau juga digunakan sebagai alat pembelajaran. Sebuah game bisa dikarakteristikan dari apa pemain lakukan misalnya :

A. Peralatan

   Misal : bola, kartu, papan, atau sebuah Komputer.

B. Peraturan

   Peraturan digunakan untuk menentukan giliran pemain, hak dan keharusan masing-masing pemain, dan tujuan permainan.

C. Skill, Strategi dan Keberuntungan

   Game dengan dengan skill, contohnya dengan kekuatan fisik, misal gulat, menembak dan kekuatan mental seperti catur.

D. Single Player Game (pemain satu orang) dan Double Player (lebih dari satu pemain)

   Jika pemain tunggal, pemain harus bermain dengan keahlian, berpacu dengan waktu dan keberuntungan sedangkan pemain double, pemain diharuskan untuk menggunakan suatu strategi dan kekompakan sesama pemain, untuk mencapai tujuan tertentu atau sebaliknya pemain harus berlomba dengan pemain lainnnya untuk mencapai sesuatu tujuan**.**

2.5.1. *Puzzle Game*

Puzzle game adalah salah satu jenis *game* yang menekankan unsur pemecahan teka-teki. Jenis-jenis teka-teki yang harus dipecahkan dapat menguji

banyak keterampilan pemecahan masalah termasuk logika, pengenalan pola, urutan pemecahan, dan penyelesaian kata.

Game puzzle fokus pada tantangan logis dan konseptual, meskipun kadang-kadang *game* menambah unsur tekanan dalam hal pembatasan waktu atau unsur lainnya. Meskipun banyak permainan *action* dan permainan petualangan melibatkan teka-teki seperti mendapatkan benda yang dapat diakses, permainan *puzzle* sejati berfokus pada pemecahan masalah sebagai kegiatan utama *gameplay puzzle*. Permainan biasanya melibatkan bentuk, warna, atau simbol, dan pemain harus secara langsung atau tidak langsung memanipulasi objek-objek game ke dalam pola tertentu.

Permainan *puzzle* biasanya menawarkan serangkaian teka-teki terkait yang adalah variasi dari satu tema. Tema ini bisa melibatkan pengenalan pola, logika, atau memahami proses. Permainan ini biasanya memiliki aturan yang sederhana, di mana pemain memanipulasi potongan permainan pada *grid*, jaringan atau ruang interaksi lainnya . Pemain harus mengungkap petunjuk untuk mencapai beberapa kondisi kemenangan, yang kemudian akan memungkinkan mereka untuk maju ke tingkat berikutnya. Melengkapi setiap teka-teki biasanya akan mengarah ke tantangan yang lebih suli , meskipun beberapa permainan menghindari melelahkan pemain dengan menawarkan tingkat lebih mudah antara yang lebih sulit.

2.6. *Tools* Yang Digunakan

Tools yang dimaksud adalah sekumpulan perangkat lunak yang membantu dalam pembuatan program dalam tugas akhir ini.

2.6.1. Javascript

JavaScript adalah bahasa skrip yang populer di internet dan dapat bekerja di sebagian besar penjelajah *web* populer seperti Internet Explorer (IE), Mozilla Firefox, Netscape dan Opera. Kode JavaScript dapat disisipkan dalam halaman web menggunakan tag SCRIPT [9].

JavaScript pertama kali dikembangkan oleh Brendan Eich dari Netscape dibawah nama Mocha, yang nantinya namanya diganti menjadi LiveScript, dan akhirnya menjadi JavaScript.

Navigator sebelumnya telah mendukung Java untuk lebih bisa dimanfaatkan para programmer yang non-Java. Maka dikembangkanlah bahasa pemrograman bernama LiveScript untuk mengakomodasi hal tersebut. Bahasa pemrograman inilah yang akhirnya berkembang dan diberi nama JavaScript, walaupun tidak ada hubungan bahasa antara Java dengan JavaScript.

JavaScript bisa digunakan untuk banyak tujuan, misalnya untuk membuat efek rollover baik di gambar maupun teks, dan yang penting juga adalah untuk membuat AJAX. JavaScript adalah bahasa yang digunakan untuk AJAX.

Kelebihan bahasa pemrograman Javascript diantaranya adalah:

1. Ukuran File Kecil

   Script dari javascript memiliki ukuran yang kecil sehingga ketika web yang memiliki javascript ditampilkan di browser maka akses tampilannya akan lebih cepat dibandingkan ketika browser membuka suatu web yang memiliki script java. Hal ini juga sangat berkepentingan dengan daya kerja server. Semakin kecil space suatu web yang disimpan dalam suatu server maka daya kerja server ketika di browsing oleh user di internet akan tidak terlalu berat, selain itu sifat javascript client side yang tidak perlu lagi di olah oleh server ketika browser memanggil web dari sebuah server.

2. Mudah dipelajari

   Javascript merupakan bahasa semi pemograman yang merupakan gabungan antara bahasa pemograman java dengan bahasa kode HTML sehingga disebut bahasa hybrid. Walaupun javascript merupakan turunan dari java namun javascript tidak memiliki aturan yang serumit java.

3. Terbuka (Open Source)

   Javascript tidak terikat oleh hardware maupun software tertentu bahkan system operasi seperti windows maupun unix. Karena ia bersifat terbuka, maka ia dapat dibuat maupun di baca di semua jenis komputer.

Sedangkan beberapa kekurangannya adalah:

1. Script tidak terenkripsi

   Karena javascript bersifat client side, maka script yang kita buat di text editor dan telah dijadikan web di server, ketika user merequest web dari server tersebut maka sintak javascript akan langsung ditampilkan di browser. User bisa melihat dan menirunya dari sourcenya.

2. Kemampuan terbatas

   Walaupun javascript mampu membuat bentuk web menjadi interaktif dan dinamis, namun javascript tidak mampu membuat program aplikasi sendiri seperti java.

3. Keterbatasan Objek

   Javascript tidak mampu membuat kelas kelas yang bisa menampung objek objek tambahan seperti java karena javascript teleh memiliki objek yang builtin pada sturktur bahasanya.

2.5.1.1. *Enchant.js*

Enchant.js merupakan sebuah kerangka kerja (*framework*) yang dibuat dalam bahasa *javascript* yang didesain untuk mengembangkan aplikasi *game* pada *platform web browser. Enchant.js* dipublikasikan pada tahun 2011 dan merupakan *framework* yang *open source,* artinya semua pihak dapat menggunakan dan mengembangkan *framework* ini.

2.5.1.2. Sizeof.js

Sizeof.js merupakan sebuah *library* dalam bahasa javascript yang dibuat oleh Stephen Morley. *Sizeof.js* berfungsi untuk mencari penggunaan memori untuk setiap objek-objek pada javascript.

2.6.2. *Web Browser*

Web Browser adalah suatu program atau software yang digunakan untuk menjelajahi internet atau untuk mencari informasi dari suatu web yang tersimpan didalam komputer. Awalnya, *web browser* berorientasi pada teks dan belum dapat

menampilkan gambar. Namun, web browser sekarang tidak hanya menampilkan gambar dan teks saja, tetapi juga memutar *file* multimedia seperti video dan suara. *Web browser* juga dapat mengirim dan menerima email, mengelola HTML, sebagai masukan dan menjadikan halaman web sebagai hasil keluaran yang informatif.

Dengan menggunakan web browser, para pengguna internet dapat mengakses berbagai informasi yang terdapat di internet dengan mudah. Beberapa contoh web browser diantaranya Internet Explorer, Mozilla, Firefox, Safari, Opera, dan sebagainya.

2.6.3. *Text Editor*

Text Editor adalah jenis program yang digunakan untuk mengolah *file* teks biasa. *Text Editor* sering disediakan dengan sistem operasi atau paket pengembangan perangkat lunak, dan dapat digunakan untuk mengubah file konfigurasi dan kode sumber bahasa pemrograman [10].

Ada perbedaan penting antara *file* teks biasa yang dibuat oleh editor teks dan *file* dokumen yang dibuat oleh pengolah kata seperti *Pages*, *Microsoft Word*, dan *WordPerfect*.

Sebuah *file* teks biasa menggunakan karakter sederhana set seperti ASCII untuk mewakili angka, huruf , dan sejumlah kecil simbol. Satu-satunya karakter non - *printing* dalam file yang dapat digunakan untuk memformat teks , adalah baris baru, tab, dan *FormFeed* .

Dokumen pengolah kata umumnya berisi teks diformat, seperti memungkinkan teks muncul di huruf tebal dan miring , untuk menggunakan beberapa *font* , dan harus terstruktur ke dalam kolom dan tabel . Kemampuan ini dulunya hanya terkait dengan *desktop publishing*, tapi sekarang tersedia dalam pengolah kata sederhana .

Halaman *web* adalah teks biasa , dengan tag HTML untuk mencapai format. Pengolah kata yang dikembangkan untuk memungkinkan format teks untuk presentasi pada halaman cetak , sementara teks yang dihasilkan oleh editor

teks umumnya digunakan untuk keperluan lain , seperti data masukan untuk program komputer .

Ketika kedua format yang tersedia, pengguna harus memilih dengan hati-hati . Menyimpan sebuah file teks biasa dalam format word processor akan menambah informasi format yang bisa mengganggu mesin - pembacaan teks . Menyimpan dokumen word processor sebagai file teks akan kehilangan informasi format .

2.6.4. *Web Server*

Server web atau pelayan web dapat merujuk baik pada perangkat keras ataupun perangkat lunak yang menyediakan layanan akses kepada pengguna melalui protokol komunikasi HTTP atau HTTPS atas berkas-berkas yang terdapat pada suatu situs web dalam layanan ke pengguna dengan menggunakan aplikasi tertentu seperti peramban web.

Penggunaan paling umum server web adalah untuk menempatkan situs web, namun pada prakteknya penggunaannya diperluas sebagai tempat peyimpanan data ataupun untuk menjalankan sejumlah aplikasi kelas bisnis.

2.6.5. OOP (*Object Oriented Programming*)

OOP (*Object Oriented Programming*) atau yang dikenal dengan Pemrograman Berorientasi Objek merupakan paradigma pemrograman yang berorientasikan kepada objek. Semua data dan fungsi di dalam paradigma ini dibungkus ke dalam kelas-kelas atau objek-objek. Model data berorientasi objek dikatakan dapat memberi fleksibilitas yang lebih, kemudahan mengubah program, dan digunakan luas dalam teknik piranti lunak skala besar. Dengan menggunakan OOP maka dalam melakukan pemecahan suatu masalah kita tidak melihat bagaimana cara menyelesaikan suatu masalah tersebut (terstruktur) tetapi objek-objek apa yang dapat melakukan pemecahan masalah tersebut.

Pemrograman orientasi-objek menekankan konsep berikut:

1. Kelas (*Class*) - kumpulan atas definisi data dan fungsi-fungsi dalam suatu unit untuk suatu tujuan tertentu.
2. Objek (*Object*) - membungkus data dan fungsi bersama menjadi suatu unit dalam sebuah program komputer. Objek merupakan dasar dari modularitas dan struktur dalam sebuah program komputer berorientasi objek.
3. Abstraksi (*Abstract*) - kemampuan sebuah program untuk melewati aspek informasi yang diproses olehnya, yaitu kemampuan untuk fokus pada inti.
4. Enkapsulasi (*Encapsulation*) - Memastikan pengguna sebuah objek tidak dapat mengganti keadaan dalam dari sebuah objek; hanya metode dalam objek tersebut yang diberi ijin untuk mengakses keadaannya.
5. Polimorfisme (*Polimorfism*) melalui pengiriman pesan.
6. Inheritas (*Inheritance*) - Mengatur polimorfisme dan enkapsulasi dengan mengijinkan objek didefinisikan dan diciptakan dengan jenis khusus dari objek yang sudah ada.

2.6.6. UML

UML (*Unified Modeling Language*) merupakan bahasa standar yang bekerja dalam *object-oriented* untuk menentukan, memvisualisasikan, merancang, dan mendokumentasikan elemen-elemen informasi yang terdapat dalam sistem software. UML mulai diperkenalkan oleh *Object Management Group*, sebuah organisasi yang telah mengembangkan model, teknologi, dan standar OOP sejak tahun 1980-an. UML merupakan dasar bagi perangkat (tool) desain berorientasi objek dari IBM [11].

Dalam UML terdapat beberapa diagram untuk memodelkan aplikasi berorientasi objek, yaitu:

1. *Use Case diagram* , untuk memodelkan proses bisnis dan merepresentasikan sebuah interaksi antara aktor dengan sistem.
2. *Class diagram* , untuk menggambarkan keadaan (atribut/properti) suatu sistem, sekaligus menggambarkan struktur dan deskripsi class, package dan objek

beserta hubungan satu sama lain seperti containment, pewarisan, asosiasi, dan lain-lain. Class memiliki tiga area pokok yakni nama (dan *stereotype*) , atribut , dan metoda .

3. *Statechart diagram* , untuk memodelkan perilaku objects di dalam sistem atau menggambarkan transisi dan perubahan keadaan (dari satu state ke state lainnya) suatu objek pada sistem. Pada umumnya statechart diagram menggambarkan class tertentu .
4. *Activity diagram* , untuk memodelkan perilaku *Use Case* dan objek di dalam sistem.
5. *Sequence diagram* , untuk memodelkan pengiriman pesan (*message*) antar objects dan juga digunakan untuk menggambarkan skenario atau rangkaian langkah-langkah yang dilakukan sebagai respons dari sebuah event untuk menghasilkan output tertentu.
6. *Collaboration diagram* , untuk memodelkan interaksi antar objects seperti sequence diagram, tetapi lebih menekankan pada peran masing-masing objek dan bukan pada waktu penyampaian message.
7. *Component diagram* , untuk memodelkan komponen objek atau menggambarkan struktur dan hubungan antar modul yang berisi kode, termasuk ketergantungan (*dependency*) di antaranya.