BAB II

LANDASAN TEORI

2.1 Algoritma

Algoritma adalah deskripsi langkah-langkah penyelesaian masalah yang tersusun secara logis atau urutan logis pengambilan keputusan untuk pemecahan suatu masalah. Algoritma ditulis dengan notasi khusus, notasi mudah dimengerti dan notasi dapat diterjemahkan menjadi sintaks suatu bahasa pemrograman (Nugraha, 2012).

Suatu algoritma akan memerlukan masukan (*input*) tertentu untuk memulainya, dan akan menghasilkan keluaran (*output*) tertentu pada akhirnya. Hal-hal yang perlu diperhatikan dalam algoritma adalah mencari langkah-langkah yang paling sesuai untuk penyelesaian suatu masalah, karena setiap algoritma memiliki karakteristik tertentu yang memiliki kelebihan dan kekurangan (Nugraha, 2012).

2.1.1 Algoritma *Shell sort*

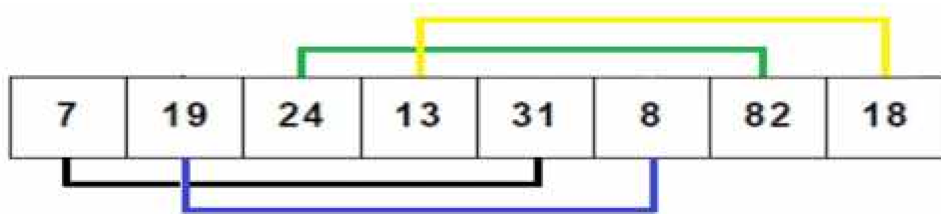
Metode ini disebut juga dengan metode pertambahan menurun (*diminishing increment*) yang ditemukan oleh tokoh bernama Donald Shell pada tahun 1959 yang kemudian populer dengan istilah *Shell Sort*. Metode ini mengurutkan data dengan cara membandingkan suatu data dengan data yang lain yang memiliki jarak tertentu, kemudian dilakukan pertukaran bila diperlukan (Warni, 2012). Proses pengurutan dengan metode *Shell sort* dapat dijelaskan sebagai berikut:

1. Gambar 1 merupakan data awal yang di acak dengan banyak data (N) adalah 8.

7	19	24	13	31	8	82	18

Gambar 2.1. Data awal (*Shell sort*)

2. Langkah pertama proses pengurutannya adalah menentukan jarak data yang akan dibandingkan yaitu dengan cara banyak data (N) dibagi 2 (8/2 = 4). Jadi, jarak perbandingan datanya adalah 4. Lihat Gambar 2.

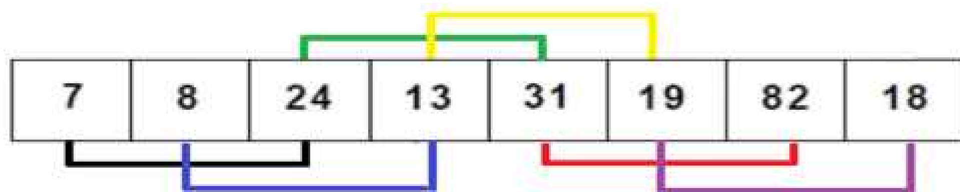


Gambar 2.2. Jarak perbandingan data langkah pertama (*Shell sort*)

Data pertama dibandingkan dengan data pada jarak ke-4 dari data pertama (7 dan 31), karena 7 lebih kecil daripada 31 maka data tidak ditukar. Kemudian data kedua dibandingkan dengan data pada jarak ke-4 dari data kedua (19 dan 8), karena 19 lebih besar dari 8, maka kedua data tersebut ditukar. Pada data ketiga dibandingkan dengan data pada jarak ke-4 dari data ketiga (24 dan 82), karena 24 lebih kecil daripada 82, maka kedua data tidak ditukar. Pada data keempat dibandingkan dengan data pada jarak ke-4 dari data keempat (13 dan 18), kedua data tidak ditukar karena 13 lebih kecil daripada 18. Hasil pengurutan sementara dari langkah pertama seperti terlihat pada Gambar 3.

Gambar 2.3 Hasil pengurutan sementara langkah pertama (*Shell sort*)

3. Pada langkah kedua digunakan jarak (N/2)/2 yaitu ((8/2)/2)=2. Jadi, jarak perbandingan datanya adalah 2. Lihat Gambar 4.



Gambar 2.4. Jarak perbandingan data langkah kedua (*Shell sort*)

Data pertama dibandingkan dengan data pada jarak ke-2 dari data pertama (7 dan 24), karena 7 lebih kecil daripada 24, maka data tidak ditukar. Pada data kedua dibandingkan dengan data pada jarak kedua dari data kedua (8 dan 13), karena 8 lebih kecil daripada 13, maka data tidak ditukar. Selanjutnya, pada data ketiga dibandingkan dengan data pada jarak ke-2 dari data ketiga (24 dan 31), karena 24 lebih kecil dari 31, maka data tidak ditukar. Pada data keempat dibandingka dengan data pada jarak ke-2 dari data keempat (13 dan 19), karena

13 kecil dari 19, maka data tidak ditukar. Pada data kelima dibandingkan dengan data pada jarak ke-2 dari data kelima (31 dan 82), karena 31 lebih kecil daripada 82, maka data tidak ditukar. Pada data keenam dibandingkan dengan data pada jarak ke-3 dari data keenam (19 dan 18), karena 19 lebih besar daripada 18, maka data ditukar. Hasil pengurutan sementara dari langkah kedua terlihat pada Gambar 5.

7	8	24	13	31	18	82	19

Gambar 2.5. Hasil pengurutan data sementara langkah kedua (*Shell sort*)

4. Langkah ketiga digunakan jarak ((N/2)/2)/2 yaitu ((8/2)/2)/2=1. Jadi, jarak perbandingan data adalah 1 yang merupakan perbandingan terakahir pada data. Lihat Gambar 6.

7	8	24	13	31	18	82	19

Gambar 2.6. Jarak perbandingan data langkah ketiga (*Shell sort*)

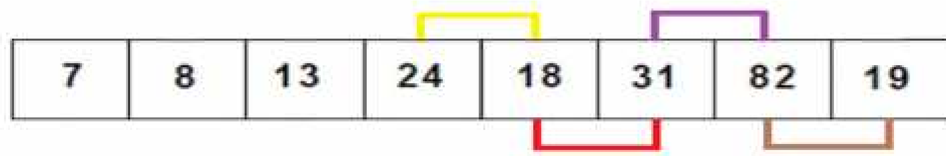
Pada Gambar 6, data pertama dibandingkan dengan data pada jarak ke-1 dari data pertama (7 dan 8), karena 7 lebih kecil daripada 8, maka data tidak ditukar. Pada data kedua dibandingkan dengan daa pada jarak ke-1 dari data kedua (8 dan 24), karena 8 lebih kecil daripada 24, maka data tidak ditukar. Pada data ketiga dibandingkan dengan data pada jarak ke-1 (24 dan 13) karena 24 lebih besar daripada 13, maka data diukar. Pada data keempat dibandingkan dengan data pada jarak ke-1 dari data keempat (24 dan 31) karena 24 lebih kecil daripada 31, maka data tidak ditukar. Lihat gambar 7.

7	8	13	24	31	18	82	19

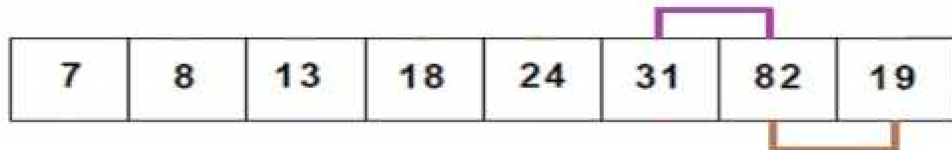
Gambar 2.7. Perbandingan data keempat (*Shell sort*)

Pada data kelima dibandingkan dengan data pada jarak ke-1 dari data kelima (31 dan 18), karena 31 lebih besar daripada 18, maka data ditukar, kemudian 18 dibandingkan lagi dengan data pada jarak ke-1 kearah kiri dari data kelima

yaitu (18 dan 24), karena 18 lebih kecil daripada 24, maka data ditukar. Lihat Gambar 2.8.

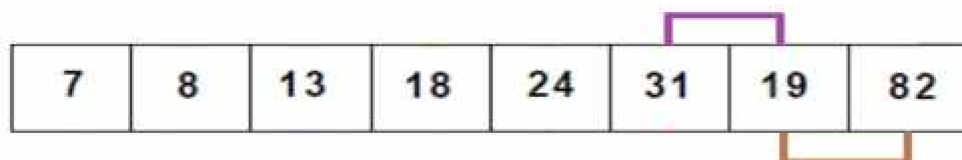


Gambar 2.8. Perbandingan data kelima (*shell sort*)



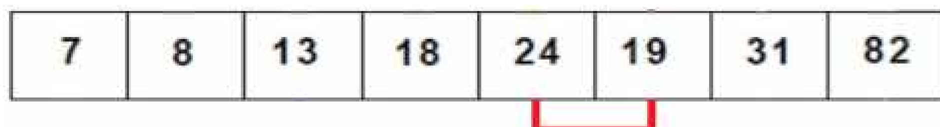
Gambar 2.9. Perbandingan data keenam (*Shell sort*)

Pada Gambar 2.9, data keenam dibandingkan dengan data pada jarak ke-1 dari data keenam (31 dan 82), karena 31 lebih kecil daripada 82, maka data tidak ditukar. Selanjutnya pada data ketujuh dibandingkan dengan data pada jarak ke-1 dari data ketujuh (82 dan 19), karena 82 lebih besar daripada 19 maka data ditukar. Lihat gambar 2.10.



Gambar 2.10. Perbandingan data ketujuh (*Shell sort*)

Kemudian 19 dibandingkan lagi dengan data pada jarak ke-1 kearah kiri dari data 19 (19 dan 31), karena 31 lebih besar daripada 19 maka data ditukar. Kemudian 19 dibandingkan lagi dengan data pada jarak ke-1 kearah kiri dari data 19 (19 dan 24), karena 24 lebih besar daripada 19 maka data ditukar Lihat Gambar 2.11.



Gambar 2.11. Perbandingan data (19 dan 24)

7	8	13	18	19	24	31	82

Gambar 2.12. Perbandingan data (19 dan 18)

Pada Gambar 2.12, selanjutnya 19 dibandingkan lagi dengan data pada jarak ke-1 dari angka 19 (19 dan 18), karena 18 lebih kecil daripada 19 berarti pada bagian data kiri dari 19 sudah tidak ada lagi data yang lebih besar dari 19, pengurutan data selesai seperti yang terlihat pada Gambar 13.

7	8	13	18	19	24	31	82

Gambar 2.13. Hasil akhir pengurutan data dengan metode *Shell sort*

Algoritma *Shell sort* lima kali lebih cepat dibandingkan algoritma pengurutan gelembung (*bubble sort*) dan dua kali lebih cepat dibandingkan algoritma pengurutan penyisipan (*insertion sort*). Algoritma *Shell sort* sangat sederhana tetapi sulit untuk di analisis secara teoritis (Durrani dkk, 2009).

Kompleksitas waktu *Shell sort* pada kondisi *best case* yaitu O(n), ini terjadi pada saat *list* data acak dalam keadaan yang sudah hampir terurut dan pada ukuran data yang sedikit. Pada kondisi *worst case* kompleksita waktu algoritma *Shell sort* menjadi $O(n^2)$, ini terjadi pada saat mengurutkan data yang besar. Pada kondisi *average case,* algoritma *Shell sort* memilki kompleksitas waktu $O(n(\log n)^2)$ (Durrani dkk, 2009).

2.1.2 Algoritma *Quick sort*

Metode *Quick sort* merupakan suatu metode yang paling cepat dalam proses pengurutan data. *Quick sort* sering disebut juga metode partisi (*partition exchange sort*). Metode ini diperkenalkan pertama kali oleh C.A.R. Hoare pada tahun 1962. Untuk mempertinggi efektifitas dari metode ini, digunakan teknik menukarkan dua elemen dengan jarak yang cukup besar (Warni, 2012).

Quick sort merupakan sebuah algoritma *sorting* dari model *divide* dan *conquer. Divide* dan *conquer* adalah metode pemecahan masalah yang bekerja

dengan membagi masalah menjadi beberapa sub-masalah yang lebih kecil, kemudian menyelesaikan masing-masing sub-masalah secara independen, dan akhirnya menggabungkannya (Durrani dkk, 2009). Berikut proses pengurutan *Quick sort* :

1. Pada Gambar 2.14 merupakan data awal yang di acak dengan banyak data (N) adalah 8.

30	70	10	40	60	20	50	80

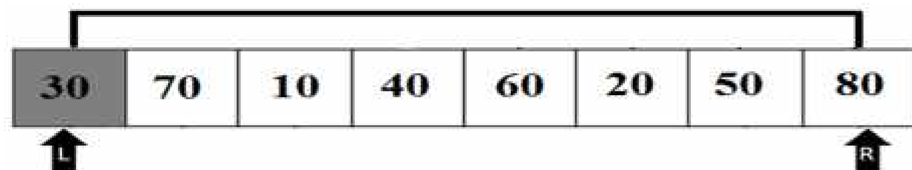
Gambar 2.14. Data awal (*Quick sort*)

2. Langkah pertama dipilih *pivot* pada data tertentu, misalnya angka 30. *Pivot* adalah nilai acuan yang dipilih untuk mengatur data di sebelah kiri *pivot* agar lebih kecil daripada *pivot* dan data di sebelah kanan *pivot* agar lebih besar daripada *pivot*. Lihat gambar 2.15.

30	70	10	40	60	20	50	80

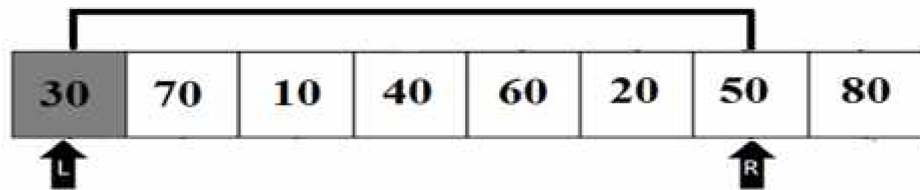
Gambar 2.15. Pemilihan *pivot* (*Quick sort*)

3. Setelah *pivot* ditentukan, langkah kedua adalah membagi data menjadi dua bagian dengan melakukan perbandingan terlebih dahulu pada data kanan (R) dan data kiri (L) dengan *pivot* yang dimulai dari data paling kanan (R). Berikut perbandingan dimulai pada data pertama dari data kanan (R) dengan *pivot*, yaitu 80 dan 30 yang terlihat pada Gambar 16.



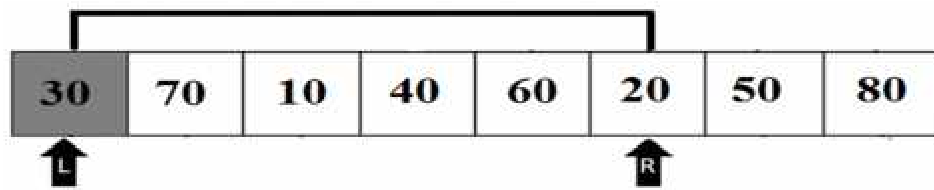
Gambar 2.16. Perbandingan data dengan *pivot* (80 dan 30)

Karena 80 lebih besar daripada 30 maka data tidak ditukar. Selanjutnya data kedua dari kanan (R) yaitu 50 dibandingkan dengan 30, karena 50 lebih besar daripada 30 maka data tidak ditukar. Lihat Gambar 2.17.

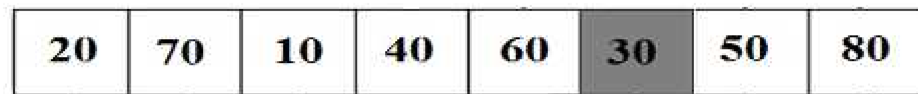


Gambar 2.17. Perbandingan data dengan *pivot* (50 dan 30)

Selanjutnya data ketiga dari kanan (R) dibandingkan dengan *pivot*, yaitu 20 dengan 30, karena 20 lebih kecil daripada 30 maka data ditukar. Lihat Gambar 2.18 dan 2.19.

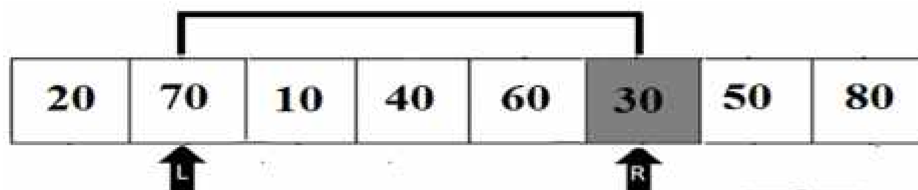


Gambar 2.18. Perbandingan data dengan *pivot* (20 dan 30)



Gambar 2.19. Pertukaran data dengan *pivot* (20 dan 30)

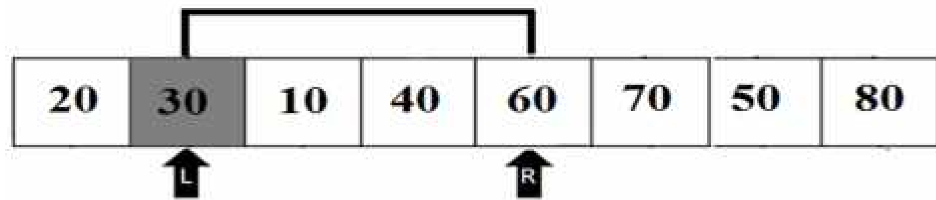
Karena terjadi pertukaran, maka perbandingan data selanjutnya dimulai dari data kiri (L) yaitu data kedua dari kiri (L) dengan *pivot*. Perbandingannya yaitu 70 dan 30, karena 70 lebih besar daripada 30 maka data ditukar. Lihat Gambar 2.20 dan 2.21.



Gambar 2.20. Perbandingan data dengan *pivot* (70 dan 30)

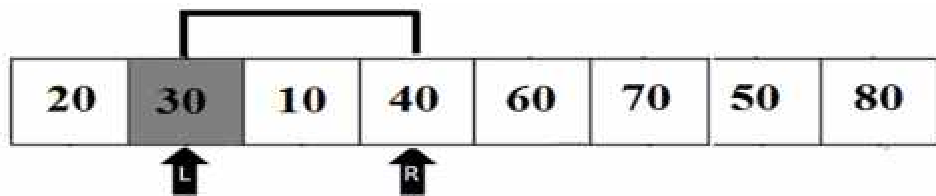
Gambar 2.21. Pertukaran data dengan *pivot* (70 dan 30)

Karena terjadi pertukaran, maka selanjutnya membandingkan data keempat dari kanan (R) dengan *pivot* (60 dan 30), karena 60 lebih besar dari 30 maka data tidak ditukar. Lihat Gambar 2.22.



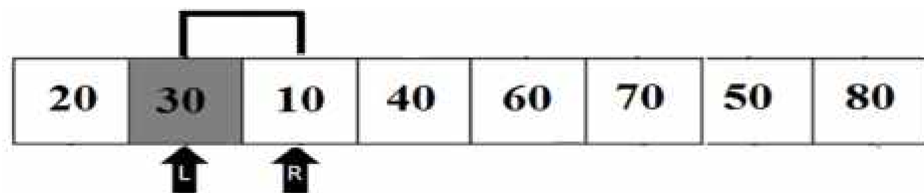
Gambar 2.22. Perbandingan data dengan *pivot* (60 dan 30)

Selanjutnya data kelima dari data kanan (R) dibandingkan dengan *pivot* (40 dan 30), karena 40 lebih besar daripada 30, maka data tidak ditukar. Lihat Gambar 2.23.



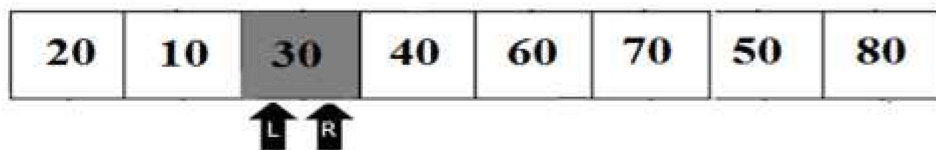
Gambar 2.23. Perbandingan data dengan *pivot* (40 dan 30)

Selanjutnya data keenam dari kanan (R) dibandingkan dengan *pivot* (10 dan 30), karena 10 lebih kecil daripada 30, maka data ditukar. Lihat Gambar 2.24.

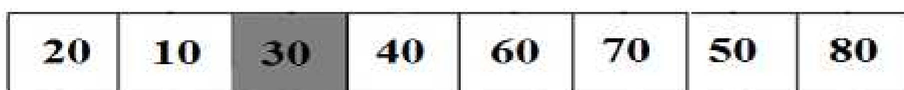


Gambar 2.24. Perbandingan data dengan *pivot* (10 dan 30)

Kemudian perbandingan data kanan (R) dan data kiri (L) bertemu pada *pivot*, perbandingan data dihentikan yang berarti data telah berada pada posisi dimana semua data kanan (R) lebih besar daripada *pivot* dan semua data kiri (L) lebih kecil daripada *pivot*, seperti yang terlihat pada Gambar 2.25 dan 2.26.



Gambar 2.25. Perbandingan data bertemu pada *pivot* (30)

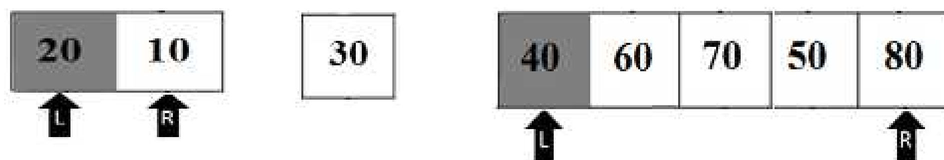


Gambar 2.26. Semua data kanan dari *pivot* lebih besar daripada *pivot* dan semua data kiri dari *pivot* lebih kecil daripada *pivot*.

Langkah selanjutnya membagi data menjadi tiga bagian yaitu tumpukan data kiri dari *pivot*, data *pivot*, dan tumpukan data kanan dari *pivot*. Lihat Gambar 2.27.

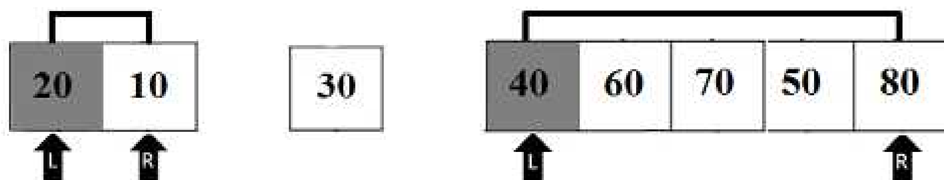
Gambar 2.27. Pembagian data

Lagkah kedua dilakukan pengurutan dengan cara sebelumnya pada tumpukan data kanan dan tumpukan data kiri dari *pivot*, yaitu dengan cara menentukan *pivot* dan membagikan data. Pada tumpukan data kiri dan data kanan dipilih *pivot* pada data pertama yaitu 20 dan 40. Sedangkan data 30 sudah tidak dilakukan perbandingan lagi. Lihat Gambar 2.28.



Gambar 2.28. Pemilihan *pivot* pada tumpukan data kiri dan data kanan

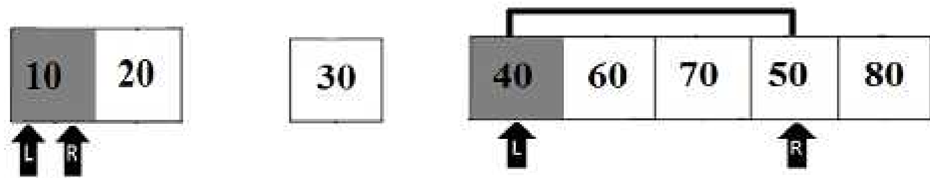
Pada tumpukan data kiri dilakukan perbandingan pada data kanan (R) dengan *pivot* (10 dan 20), karena 10 lebih kecil daripada 20 maka data ditukar, dan data tidak ada yang dibandingkan lagi. Kemudian pada tumpukan data kanan dilakukan perbandingan pada data pertama dari kanan (R) dengan *pivot* (80 dan 40), karena 80 lebih besar daripada 40, maka data tidak ditukar. Lihat Gambar 2.29.



Gambar 2.29. Perbandingan data dengan *pivot* pada tumpukan data kiri (10 dan 20) dan data kanan (80 dan 40)

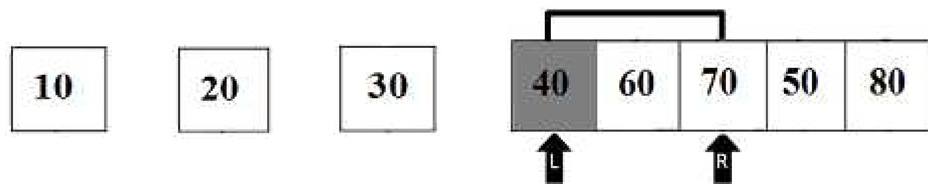
Semua data pada tumpukan kiri telah dibandingkan. Selanjutnya pada tumpukan data kanan, perbandingan dimulai pada data kedua dari kanan (R)

dengan *pivot* (50 dan 40), karena 50 lebih besar daripada 40 maka data tidak ditukar. Lihat Gambar 2.30.



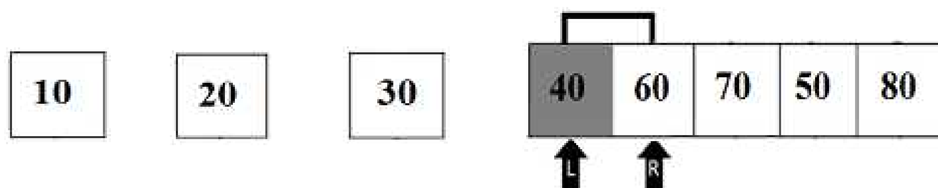
Gambar 2.30. Perbandingan data dengan *pivot* pada tumpukan data kanan (50 dan 40)

Pada tumpukan data kiri yaitu 10 dan 20 dipisahkan karena sudah tidak dilakukan perbandingan lagi. Pada tumpukan data kanan dilakukan perbandingan pada data ketiga dari kanan (R) dengan *pivot* (70 dan 40), karena 70 lebih besar daripada 40 maka data tidak ditukar. Lihat Gambar 2.31.

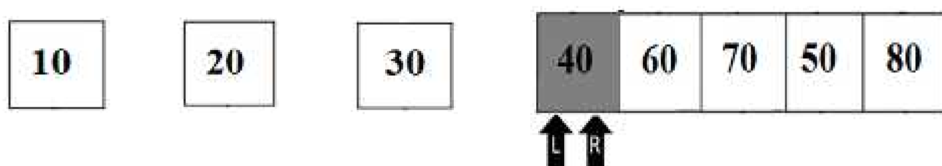


Gambar 2.31. Perbandingan data dengan *pivot* (70 dan 40)

Selanjutnya data keempat dari kanan (R) dengan *pivot* (60 dan 40), karena 60 lebih besar daripada 40 maka data tidak ditukar. Kemudian tidak ada data yang dibandingkan lagi dengan *pivot.* Lihat Gambar 2.32 dan 2.33.



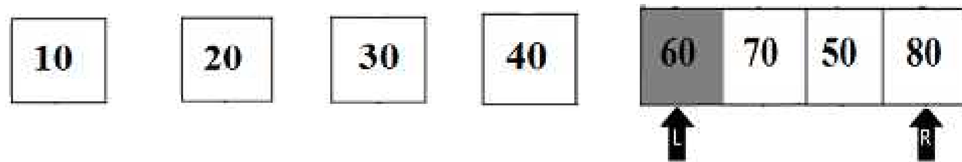
Gambar 2.32. Perbandingan data dengan *pivot* (60 dan 40)



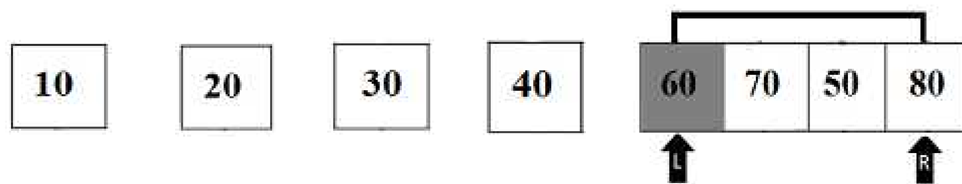
Gambar 2.33. Perbandingan bertemu pada *pivot* (40)

Selanjutnya karena perbandingan data pada tumpukan kanan dengan *pivot* telah selesai, maka data 40 dipisahkan. Kemudian dipilih *pivot* pada data pertama

dari tumpukan data kanan, yaitu 60. Kemudian dilakukan perbandingan pada data pertama dari kanan (R) dengan *pivot* (80 dan 60), karena 80 lebih besar daripada 60 maka data tidak ditukar. Lihat Gambar 2.34 dan 2.35.

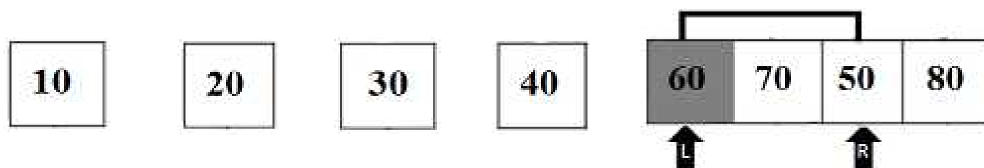


Gambar 2.34. Pemilihan *pivot*



Gambar 2.35. Perbandingan data dengan *pivot* (80 dan 60)

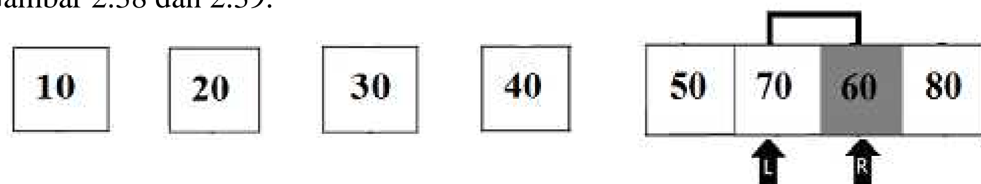
Selanjutnya perbandingan dilakukan pada data kedua dari kanan (R) dengan *pivot* yaitu 50 dan 60, karena 50 lebih kecil daripada 60 maka data ditukar. Lihat Gambar 2.36 dan 2.37.



Gambar 2.36. Perbandingan data dengan *pivot* (50 dan 60)

Gambar 2.37. Pertukaran data dengan *pivot* (50 dan 60)

Selanjutnya, karena telah terjadi pertukaran, maka perbandingan data dimulai dari data kiri (L) yaitu data kedua dari data kiri (L) dibandingkan dengan *pivot* (70 dan 60), karena 70 lebih besar daripada 60 maka data ditukar. Lihat Gambar 2.38 dan 2.39.



Gambar 2.38. Perbandingan data dengan *pivot* (70 dan 60)

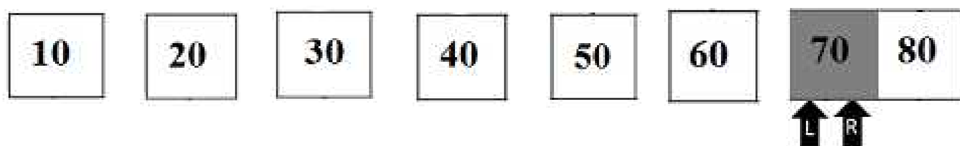
Gambar 2.39. Pertukaran data dengan *pivot* (70 dan 60)

Selanjutnya membagi tumpukan data menjadi tiga bagian yaitu bagian tumpukan kiri dari *pivot*, *pivot*, dan bagian kanan dari *pivot*. Lihat Gambar 40.

Gambar 2.40. Pembagian data

Pada tumpukan data kiri dan juga *pivot* tidak ada data yang dibandingkan lagi. Pada tumpukan data kanan dipilih *pivot* pada data pertama dari kiri data (L) yaitu 70. Kemudian melakukan perbandingan pada data pertama dari data kiri (L) dengan *pivot* (80 dan 70), karena 80 lebih besar dari 70 maka data tidak ditukar. Kemudian kedua data dipisahkan. Lihat Gambar 2.41, 2.42 dan 2.43.

Gambar 2.41. Perbandingan data dengan *pivot* (80 dan 70)



Gambra 2.42. Pertukaran data dengan *pivot* (80 dan 70)

Gambar 2.43. Hasil dari semua data yang terpisahkan

Karena sudah tidak ada lagi data yang dibandingkan, maka data yang terpisah digabungkan kembali yang telah siap terurut seperti yang terlihat pada Gambar 2.44.

10	20	30	40	50	60	70	80

Gambar 2.44. Hasil pengurutan dengan metode *Quick sort*

Kompleksitas waktu *Quick sort* pada kondisi *best case* dan *average case* adalah sama, yaitu O(nlog n). Pada kondisi *worst case* kompleksitas waktunya menjadi $O(n^2)$, ini terjadi apabila data yang akan diurutkan sudah dalam keadaan urut terbalik (Durrani dkk, 2009).

2.2 Kompleksitas Waktu

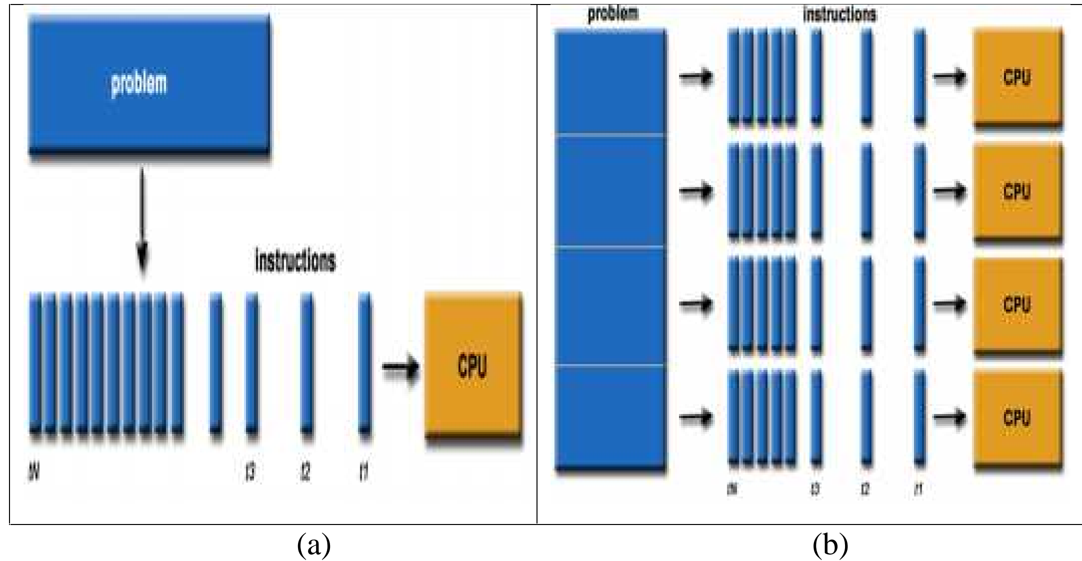
Sebuah algoritma tidak saja harus menghasilkan keluaran yang benar, tetapi juga harus efisien. Kebenaran suatu algoritma harus diuji dengan jumlah masukan tertentu untuk melihat kinerja algoritma berupa waktu yang diperlukan untuk menjalankan algoritmanya. Algoritma yang bagus adalah algoritma yang efisien. Keefisien algoritma diukur dari jumlah waktu yang dibutuhkan untuk menjalankan algoritma tersebut. Algoritma yang efisien adalah algoritma yang meminimumkan kebutuhan waktu (Nugraha, 2012).

Kompleksitas waktu dari algoritma adalah mengukur jumlah perhitungan komputasi yang dikerjakan oleh komputer ketika menyelesaikan suatu masalah dengan menggunakan algoritma. Ukuran yang dimaksud mengacu ke jumlah langkah-langkah perhitungan dan waktu tempuh pemrosesan. Kompleksitas waktu merupakan hal penting untuk mengukur efisiensi suatu algoritma. Kompleksitas waktu dari suatu algoritma yang terukur sebagai suatu fungsi ukuran masalah. Kompleksitas waktu dari algoritma berisi ekspresi bilangan dan jumlah langkah yang dibutuhkan sebagai fungsi dari ukuran permasalahan (Nugraha, 2012).

2.3 Pemrograman Paralel

Pemrograman paralel adalah teknik pemrograman komputer yang memungkinkan eksekusi perintah/operasi secara bersamaan (komputasi paralel), baik dalam komputer dengan satu (prosesor tunggal) ataupun banyak (prosesor ganda dengan mesin paralel) CPU. Pada komputasi serial, permasalahan diselesaikan dengan serangkaian instruksi yang dieksekusi satu demi satu oleh CPU, dimana hanya satu instruksi yang bisa berjalan pada satu waktu saja. Hal ini akan membutuhkan waktu eksekusi yang panjang

dan membutuhkan sumber daya komputasi yang besar pada prosesor dan memori (Syaputra dan Akbar, 2011). Gambar perbandingan antara komputasi serial dengan komputasi paralel seperti terlihat pada Gambar 2.45.



(a) (b)

Gambar 2.45 (a) Komputasi Tunggal/Serial, (b) Komputasi Paralel

(Sumber : Syaputra dan Akbar, 2011)

Dari gambar di atas terlihat bahwa kinerja komputasi paralel lebih efektif dan dapat menghemat waktu bila dibandingkan dengan komputasi serial, karena permasalahan pada komputasi paralel diselesaikan secara bersamaan. Sedangkan pada komputasi serial permasalah diselesaikan secara satu persatu. Supaya kemampuan komputer paralel dapat dimanfaatkan sepenuhnya maka diperlukan program yang secara khusus ditulis untuk komputer paralel. Dalam pembuatan program khusus tersebut digunakan bahasa pemrograman paralel beserta kompilatornya (Syaputra dan Akbar 2011).

Selain pembuatan program dan bahasa pemrograman, hal penting yang perlu diketahui yaitu parameter untuk mengukur kinerja pada komputasi paralel adalah waktu eksekusi dan *speed up*. Waktu eksekusi dapat diartikan sebagai waktu berlangsungnya (*running*) program pada arsitektur komputer. *Speed up* dari suatu program paralel adalah waktu eksekusi sekuensial dibagi dengan waktu eksekusi paralel (Syaputra dan Akbar 2011). Untuk menghitung nilai *speed up* dapat dinyatakan dengan persamaan (Wilkinson dan Allen, 2010):

$$S_p = \frac{T_s}{T_p} \quad (2.1)$$

S_p = *Speed up*

T_s= Waktu eksekusi menggunakan satu prosesor (algoritma serial)

T_p= Waktu eksekusi menggunakan p prosesor (algoritma paralel)

Speed up pada satu prosesor adalah sama dengan satu, dan *speed up* pada p prosesor bernilai 1 $S_p \leq p$. Secara ideal *speed up* meningkat sebanding dengan bertambahnya jumlah prosesor. Jadi jika digunakan p prosesor, *speed up* idealnya adalah p. Selain nilai *speed up*, sebuah sistem pararel sering kali dievaluasi dengan nilai efisiensi. Efisiensi merupakan indikator atas tingkat kinerja *speed up* yang dicapai dibandingkan dengan nilai maksimum yang dapat dicapai. Nilai kisaran efisiensi antara $\frac{1}{p} \leq E \leq 1$ (Kartawidjaja, 2008). Secara matematis efisiensi dinyatakan pada persaman (Wilkinson dan Allen, 2010):

$$\boldsymbol{E} = \frac{S_p}{p} \text{ x } 100\% \tag{2.2}$$

E = Efisiensi

S_p= *Speed up*

p = Jumlah prosesor yang digunakan

2.4 *Message Passing Interface* (MPI)

Message Passing Interface (MPI) adalah sebuah standar pemrograman yang memungkinkan pemrogram untuk membuat sebuah aplikasi yang dapat dijalankan secara paralel dengan spesifikasi *library* pemrograman untuk meneruskan pesan (*message-passing*), yang diajukan sebagai standar oleh berbagai komite dari vendor, pelaksana dan pemakai. MPI menyediakan fungsi-fungsi untuk menukarkan antar pesan (Suhartanto, 2006).

Pada *message passing,* salah satu komputer akan bertindak sebagai *master* dengan tugas utama sebagai manajer dan lainnya sebagai *slave* dengan tugas melakukan komputasi sesuai arahan *master*. Karena masing-masing pemroses tersebut harus turut melakukan komputasi, maka interaksi pengiriman data dari satu pemroses ke pemroses lainnya dilakukan dengan cara pertukaran data/pesan (*message passing*) (Suhartanto, 2006). Model *message passing* dapat dilihat pada Gambar 2.46.

Gambar 2.46. Model *Massage Passing*

(Sumber : Suhartanto, 2006)

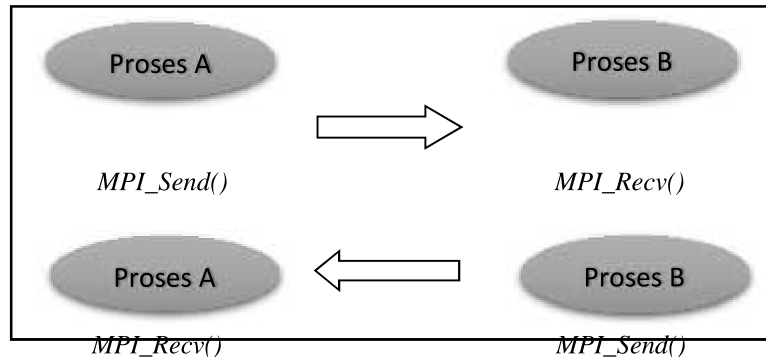
Dengan model *message passing* memungkinkan sekumpulan komputer yang heterogen terlibat didalam satu sistem komputer. Teknologi yang menerapkan paradigma *message passing* yaitu pada komputer *cluster*, dimana beberapa komputer digabungkan kedalam satu sistem oleh suatu sistem operasi dan menggunakan teknologi jaringan. Komputer tersebut bisa berupa *workstation*, *multiprocessor*, *specialized graphic engine* sampai dengan *vector supercomputer* yang dihubungkan dengan jaringan (Syaputra dan Akbar, 2011).

Pertengahan tahun 90-an muncul suatu konsorsium yang mencoba membuat standarisasi untuk teknologi perangkat lunak pertukaran pesan pada MPI, yang kemudian telah berhasil membuat standarisasi MPI-1 menggunakan bahasa pemrograman standar yaitu bahasa C dan Fortran-77, kemudian dilanjutkan tahun 1995 dengan standarisasi MPI-2 yang meliputi bahasa pemrograman standar C, C++, Fortran-77 dan Fortran-90, serta MATLAB pada tahun 2001. Pada tahun 2012 muncul standarisasi MPI-3 yang hanya meliputi tiga bahasa pemrograman yaitu Fotran-90, Fortran 2008 dan C (Forum MPI, 2012). MPI telah diimplementasikan oleh beberapa pihak pengembang dan institusi tertentu dalam teknologi komputasi paralel sehingga muncul beberapa versi MPI seperti LAM-MPI, MPICH dan OpenMPI (Syaputra dan Akbar, 2011).

Dalam implementasinya MPI menggunakan fungsi-fungsi pustaka yang dapat dipanggil dari program C, C++, Fortran-77, dan Fortran-90 untuk menjalankannya secara paralel dengan memanggil rutin *library* yang sesuai (Suhartanto, 2006). Namun, bahasa pemrograman C++ telah dihapus dari standarisasi MPI-3 dan diganti dengan bahasa pemograman Fortran 2008 (Forum MPI, 2012).

Mekanisme dasar dari sistem komunikasi pada MPI adalah proses pertukaran data pada sepasang proses dimana satu sebagai pengirim dan satunya lagi sebagai penerima. Hampir sebagian besar komunikasi yang terjadi pada MPI didasarkan pada komunikasi

point to point, sehingga komunikasi ini sangatlah penting dan sebagai dasar untuk komunikasi pada MPI (Kurniawan, 2010). Komunikasi *point to point* seperti terlihat pada Gambar 2.47.



Gambar 2.47. Komunikasi *point to point*

(Sumber : Kurniawan, 2010)

Dari Gambar 2.47 terlihat bahwa proses A mengirim pesan ke proses B, kemudian proses B kembali mengirim pesan ke proses A. Kedua proses tersebut saling melakukan pertukaran pesan dengan saling mengirim dan menerima data diantara keduanya. Pada MPI ada beberapa fungsi yang disediakan untuk mengirim dan menerima data baik secara *blocking* maupun *nonblocking* (Kurniawan, 2010).

Pada penelitian ini digunakan fungsi operasi *nonblocking*. Operasi *Nonblocking send/receive* adalah proses yang akan mengembalikan suatu nilai meskipun *buffer* belum penuh dengan data yang akan dikirim/diterima, artinya *nonblocking send/receive* akan mengembalikan nilai walaupun data yang dikirim/diterima belum dieksekusi. Operasi *nonblocking* MPI dikenali dengan nama operasinya dimana dapat dikategorikan menjadi empat bagian yaitu (Kurniawan, 2010):

a. *Immediate,* disingkat dengan I atau i
b. *Buffer,* disingkat dengan B atau b
c. *Synchronous,* disingkat dengan S atau s
d. *Ready,* disingkat dengan R atau r

Pada penelitian ini operasi *nonblocking* yang digunakan adalah *nonblocking send/recv immediate.* Pada operasi *MPI_Isend()* dan *MPI_Irecv,* karakter I menunjukkan operasi MPI dengan mode segera (*Immediate*). Operasi MPI_*Isend()* dan *MPI_Irecv* merupakan operasi untuk pengiriman dan penerimaan data secara *nonblocking* dan bersifat

segera (*immediate*) (Kurniawan, 2010). Deklarasi kode *nonblocking send/recv* mode *immediate* seperti terlihat pada Tabel 2.1.

Kode	Bahasa C
Send	int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
Recv	int MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

Tabel 2.1. Deklarasi kode *nonblocking send/recv* mode *immediate*

Sumber : Kurniawan (2010)

Dari deklarasi kode pada Tabel 2.1 terdapat beberapa fungsi dan keterangan parameter *nonblocking send/recv immediate* seperti terlihat pada Tabel 2.2 dan 2.3.

Tabel 2.2. Keterangan parameter *nonblocking send immediate*

Parameter	Fungsi	Keterangan
Buf	*Input*	*Buffer* data yang akan dikirim
Count	*Input*	Jumlah *buffer* data
Datatype	*Input*	Tipe data dari data yang akan dikirim
Dest	*Input*	Tujuan *rank* yang akan dikirim
Tag	*Input*	*Message tag* yang nilainya antara 0 sampai 32767
Comm	*Input*	*Communicator* yang digunakan
Request	*Output*	*Output* dari komunikasi

Sumber : Forum MPI (2012)

Tabel 2.3. Keterangan parameter *nonblocking recv immediate*

Parameter	Fungsi	Keterangan
Buf	*Output*	*Buffer* data yang akan diterima
Count	*Input*	Jumlah *buffer* data
Datatype	*Input*	tipe data dari data yang akan diterima
Source	*Input*	Sumber *rank* yang akan ditunggu data yang masuk

Tag	*Input*	*Message tag* yang nilainya antara 0 sampai 32767
Comm	*Input*	*Communicator* yang digunakan
Request	*Output*	*Output* dari komunikasi

Sumber : Forum MPI (2012)

2.5 Jaringan Komputer

Jaringan komputer merupakan sekumpulan komputer yang terhubung bersama dan dapat berbagi sumber daya yang dimilikinya, seperti printer, CDROM, pertukaran *file*, dan komunikasi secara elektronik antar komputer. Hubungan antar komputer dalam jaringan dapat menggunakan media kabel, telepon, gelombang radio, satelit atau sinar infra merah (*infrared*) (Zulkarnain dan Saripurna, 2012)

LAN (*Local Area Network*) adalah jaringan milik pribadi di dalam sebuah gedung atau kampus yang berukuran sampai beberapa kilometer. LAN seringkali digunakan untuk menghubungkan komputer-komputer pribadi dan *workstation* dalam kantor perusahaan atau pabrik-pabrik untuk pemakaian bersama *resource* (misalnya *printer, scanner*) dan saling bertukar informasi (Zulkarnain dan Saripurna, 2012).

Jaringan kerja *point to point* merupakan jaringan kerja yang paling sederhana tetapi dapat digunakan secara luas. Begitu sederhananya jaringan ini, sehingga seringkali tidak dianggap sebagai suatu jaringan tetapi hanya merupakan komunikasi biasa (Zulkarnain dan Saripurna, 2012). Jaringan kerja *point to point* seperti terlihat pada Gambar 2.48.

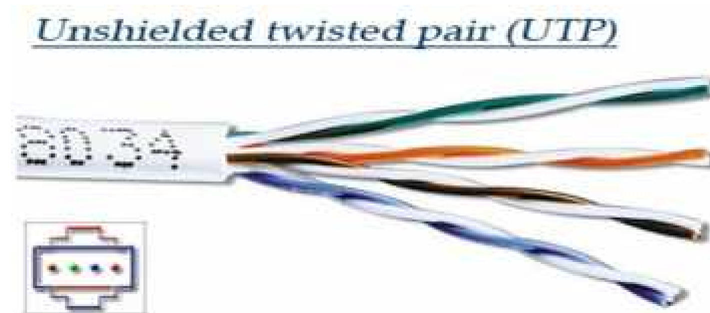
Gambar 2.48. Topologi jaringan *point to point*

(Sumber : Zulkarnain dan Saripurna, 2012)

Dari Gambar 2.48, terlihat kedua simpul mempunyai kedudukan yang setingkat, sehingga simpul manapun dapat memulai dan mengendalikan hubungan dalam jaringan tersebut. Data dikirim dari satu simpul langsung ke simpul lainnya sebagai penerima, misalnya antara terminal dengan CPU (Zulkarnain dan Saripurna, 2012).

2.6 Konfigurasi Kabel LAN / UTP (*Unshielded Twisted Pair*)

Kabel UTP merupakan salah satu media transmisi yang paling banyak digunakan untuk membuat sebuah jaringan local (*Local Area Network*), selain karena harganya relatif murah, mudah dipasang dan cukup bisa diandalkan. Sesuai namanya, *Unshielded Twisted Pair* berarti kabel pasangan berpilin/terbelit (*twisted pair*) tanpa pelindung (*unshielded*). Fungsi lilitan ini adalah sebagai eliminasi terhadap induksi dan kebocoran. Contoh kabel UTP dapat dilihat pada Gambar 2.49 (Yudianto, 2007).



Gambar 2.49. Kabel UTP (*Unshielded Twisted Pair*)

(Sumber : Yudianto, 2007)

Untuk pemasangan kabel UTP, terdapat dua jenis pemasangan kabel UTP yang umum digunakan pada jaringan komputer terutama LAN, yaitu *Straight Through Cable* dan *Cross Over Cable.*

Penggunaan kabel *cross over* adalah untuk komunikasi antar komputer (langsung tanpa HUB), atau dapat juga digunakan untuk meng-*cascade* HUB jika diperlukan. Sekarang ini ada beberapa jenis HUB yang dapat di-*cascade* tanpa harus menggunakan kabel *cross over*, tetapi juga dapat menggunakan kabel *straight through*. Kabel *cross over* menggunakan EIA/TIA 568A pada salah satu ujung kabelnya dan EIA/TIA 568B pada ujung kabel lainnya. Konfigurasi kabel *cross over* seperti terlihat pada Gambar 2.52 (Yudianto, 2007).

Gambar 2.52. Susunan kabel *cross over*

(Sumber : Yudianto, 2007)

Pada gambar di atas, pin 1 dan 2 di ujung A terhubung ke pin 3 dan 6 di ujung B, begitu pula pin 1 dan 2 di ujung B terhubung ke pin 3 dan 6 di ujung A. Jadi, pin 1 dan 2 pada setiap ujung kabel digunakan untuk mengirim data, sedangkan pin 3 dan 6 pada setiap ujung kabel digunakan untuk menerima data, karena pin 1 dan 2 saling terhubung secara berseberangan dengan pin 3 dan 6. Untuk mengenali sebuah kabel apakah *cross over* ataupun *straight* adalah dengan hanya melihat salah satu ujung kabel. Jika urutan warna kabel pada pin 1 adalah putih hijau, maka kabel tersebut adalah kabel *cross over* (padahal jika ujung yang satunya lagi juga memiliki urutan warna yang sama yaitu putih hijau sebagai pin 1, maka kabel tersebut adalah kabel *straight*). Tapi untungnya, kebanyakan kabel menggunakan standar EIA/TIA 568B pada kedua ujung kabelnya (Yudianto, 2007).

Manfaat kabel *cross over* :

1. Menghubungkan 2 buah komputer secara langsung.
2. Menghubungkan 2 buah HUB/*switch* menggunakan *port* biasa di antara kedua HUB/*switch*.
3. Menghubungkan komputer ke *port uplink switch*.
4. Menghubungkan *port* LAN *router* ke *port* biasa di HUB/*switch*.