Pendahuluan PAA 1

Wijayanti n khotimah, m.sc.

Tujuan Perkuliahan

- Mahasiswa dapat **menjelaskan** peranan algoritma, **merepresentasikan** algoritma, **merepresentasikan** algoritma ke dalam bentuk pseudocode.

Agenda

- Peranan algoritma
- Problem solving dengan algoritma
- Jenis-jenis problem komputasi

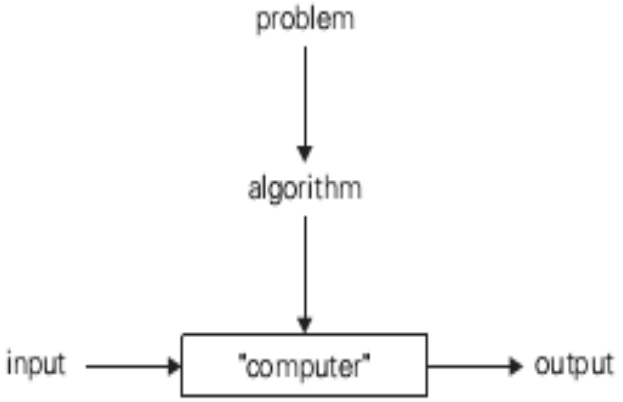
PERANAN ALGORITMA

Definisi Algoritma

- Prosedur komputasional yang didefinisikan dengan baik yang mengambil masukan sebuah nilai, atau sekumpulan nilai sebagai ***input*** dan menghasilkan sebuah nilai, atau sekumpulan nilai sebagai ***output***

- Urutan langkah-langkah komputasional yang mentransformasi *input* menjadi *output*

- Sebuah perangkat untuk memecahkan sebuah permasalahan komputasional (***computational problem***) dengan spesifikasi tertentu



- Sorting
- Searching
- Shortest paths in a graph
- Minimum spanning tree
- Traveling salesman problem
- Knapsack problem
- Dan lain-lain

Contoh computational problem: sorting

- Statement of problem:
  - *Input:* A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$
  - *Output:* A reordering of the input sequence $\langle a'_1, a'_2, \ldots, a'_n \rangle$ so that $a'_i \leq a'_j$ whenever $i < j$
- Instance: The sequence <5, 3, 2, 8, 3>
- Algorithms:
  - Selection sort
  - Insertion sort
  - Merge sort
  - (many others)

Sejarah Algoritma

- Muhammad ibn Musa al-Khwarizmi – abad ke 9 (Seorang ahli matematika)
  www.lib.virginia.edu/science/parshall/khwariz.html

Mengapa Belajar Algoritma?

- Theoretical importance
  - the core of computer science
- Practical importance
  - A practitioner's toolkit of known algorithms
  - Framework for designing and analyzing algorithms for new problems

Example: Google's PageRank Technology, Google Map

Beberapa Fakta tentang Algoritma

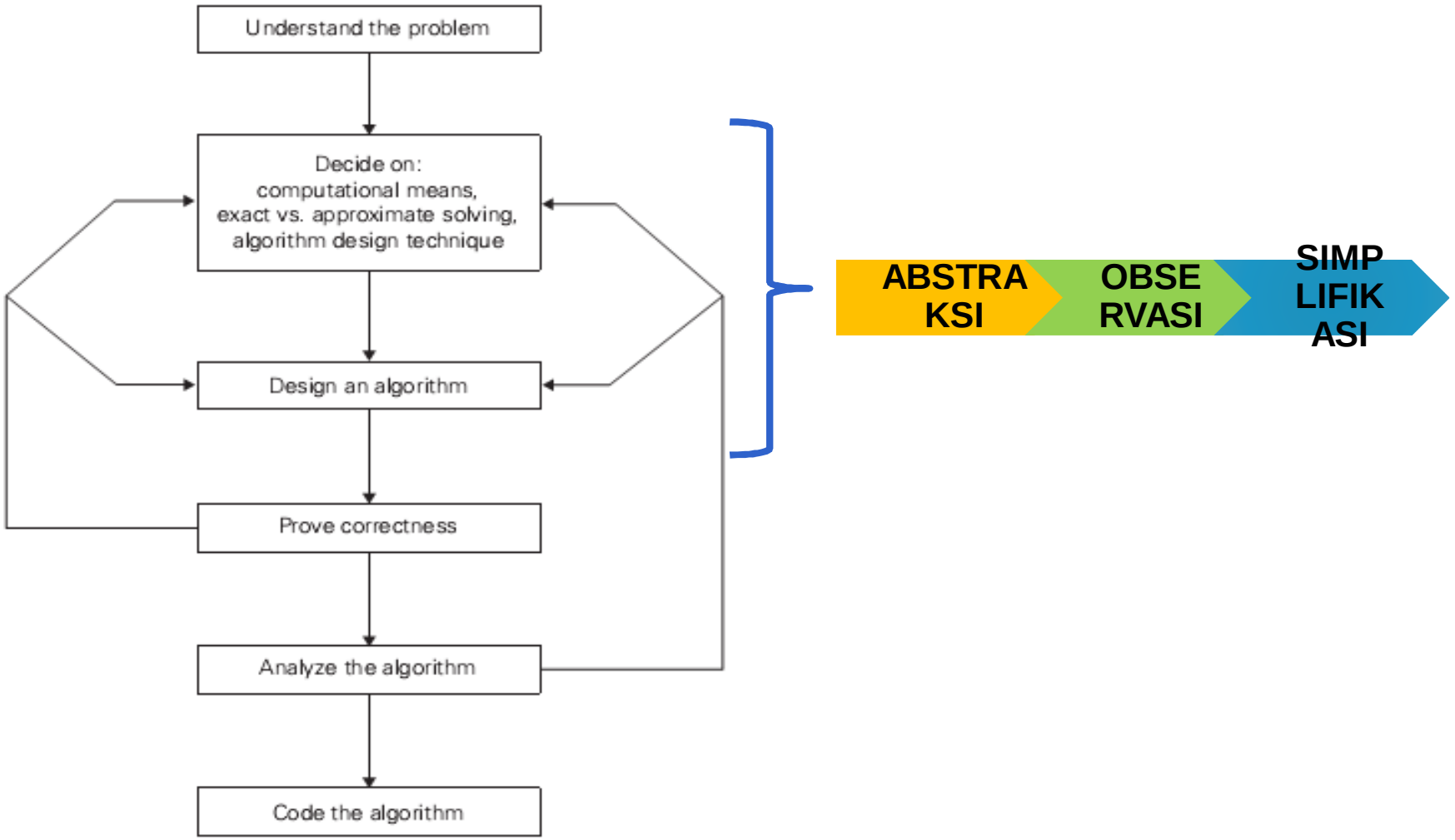
- Setiap langkah di dalam algoritma tidak boleh ada yang ambigu (harus jelas).
- Rentang input untuk algoritma harus diperhatikan.
- Sebuah algiritma dapat direpresentasikan dengan cara yang berbeda-beda.
- Kemungkinan ada beberapa algoritma untuk menyelesaikan permasalahan yang sama.

Dua Isu Utama Mengenai Algoritma

- How to design algorithms
- How to analyze algorithm efficiency

Langkah-langkah untuk Mendesain dan Menganalisis Algoritma

PROBLEM SOLVING DENGAN ALGORITMA


Understand the problem
Decide on:
computational means,
exact vs. approximate solving,
algorithm design technique
Design an algorithm
Prove correctness
Analyze the algorithm
Code the algorithm
ABSTRA
KSI
OBSE
RVASI
SIMP
LIFIK
ASI

Langkah 1: Memahami Permasalahan (Problem)

Untuk bisa memahami permasalahan (problem dengan mudah), mahasiswa harus terbiasa menyelesaikan problem

Langkah 2: Pilih antara Exact atau Approximate Problem Solving

Ada beberapa kondisi suatu permasalahan hanya bisa diselesaikan dengan pendekatan:

1. Beberapa input dari problem tersebut tidak bisa diselesaikan dengan angka pasti. Contoh: menghitung akar, persamaan non linear.
2. Problem tersebut terlalu kompleks sehingga jika mencari solusi pastinya sangat lama. Contoh: optimasi

Langkah 3: Mendesain Algoritma

- Pilih struktur data yang cocok.
- Representasi algoritma bisa dalam bentuk pseudocode atau flowchart

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Langkah 3: Mendesain Algoritma

- Pendekatan Iteratif
  - Misalnya: *Insertion Sort*
    - ✓Dengan subarray A[1 .. j-1] yang sudah terurut, masukkan elemen A[j] pada tempat yang sesuai, menghasilkan subarray A[1 .. j] yang terurut.

- Pendekatan *Divide-and-Conquer*
  - Misalnya: *Merge Sort*
    - ✓Memecah permasalahan menjadi beberapa sub-permasalahan yang serupa dengan permasalahan aslinya tapi dengan ukuran lebih kecil, memecahkan sub-permasalahan secara rekursif, lalu menggabungkan solusi-solusinya untuk menghasilkan sebuah solusi untuk permasalahan aslinya.

Langkah 4: Buktikan Kebenaranya (Correctness)

- Teknik yang paling umum untuk membuktikan kebenaran suatu algoritma adalah dengan menggunakan **induksi matematika** pada algoritma rekursive.
- Teknik lainnya adalah dengan menggunakan **looping invariant** pada algoritma iteratif.
- Membuktikan kebenaran suatu algoritma tidak cukup hanya menggunakan sebuah input tetapi membuktikan bahwa suatu algoritma salah, cukup menggunakan satu input.

Langkah 5: Menganalisa Algoritma

- Definisi: memprediksi sumber daya yang dibutuhkan algoritma
- How good is the algorithm?
  - Correctness
  - Time efficiency
  - Space efficiency
- Does there exist a better algorithm?
  - Lower bounds
  - Optimality

Langkah 5: Menganalisa Algoritma

- Menentukan jumlah “waktu” yang dibutuhkan sebuah algoritma untuk dieksekusi.
- “waktu” disini **bukan** jumlah detik atau ukuran waktu sebenarnya, tetapi sebuah aproksimasi jumlah operasi yang dilakukan sebuah algoritma.
- Baik buruknya algoritma **bukan** bergantung pada waktu eksekusi yang sebenarnya.
- Selanjutnya, penyebutan waktu mengacu pada “waktu”, bukan waktu yang sebenarnya.

Langkah 5: Menganalisa Algoritma

- Waktu yang dibutuhkansebuah algoritma berkembang sejalan dengan ukuran input.
- Waktu eksekusi (running time) sebuah program digambarkan sebagai sebuah fungsi terhadap ukuran input.
- ***Ukuran input (input size)***
  - ✓ *Misalnya jumlah item-item dalam sebuah input*
  - ✓ Jika input sebuah algoritma adalah sebuah graph, ukuran input-nya bisa jumlah verteks dan edge dalam graph.
- ***Waktu eksekusi (running time)***
  - ✓ Jumlah operasi primitif atau "step" yang dieksekusi
  - ✓ Sebuah waktu konstan dibutuhkan untuk mengeksekusi setiap baris dalam *pseudocode*.

Langkah 6: Mengimplementasikan Algoritma

- Proses dimana anda mengimplementasikan desain algoritma ke dalam baris-baris code di bahasa pemrograman seperti C, Java, Matlab, dan lain-lain

Hasil yang Diharapkan

Sebuah algoritma yang baik dengan kriteria

- **Benar (*Correct*)**
  - Jika untuk semua input, algoritma mampu menghasilkan output yang benar
  - Algoritma yang benar mampu memecahkan (***solves***) permasalahan komputasional

- **Efisien**
  - Beberapa algoritma yang dirancang untuk memecahkan permasalahan yang sama seringkali memiliki tingkat efisiensi yang sangat berbeda.

Mengapa Butuh Algoritma yang Baik?

- Komputer bisa saja cepat, tetapi kecepatannya tidak tak terbatas.
- Memori mungkin saja tidak mahal tetapi tidak gratis.
- Komputasi membutuhkan sumber daya dan space di dalam memori. Oleh karena itu, kita harus menggunakan sumber daya tersebut dengan bijak.
- Algoritma yang efisien baik secara waktu maupun space dapat membantu.

Contoh:

As an example, in Chapter 2, we will see two algorithms for sorting. The first, known as ***insertion sort***, takes time roughly equal to $c_1 n^2$ to sort n items, where c_1 is a constant that does not depend on n. That is, it takes time roughly proportional to n^2. The second, ***merge sort***, takes time roughly equal to $c_2 n \lg n$, where $\lg n$ stands for $\log_2 n$ and c_2 is another constant that also does not depend on n. Insertion sort typically has a smaller constant factor than merge sort, so that $c_1 < c_2$. We shall see that the constant factors can have far less of an impact on the running time than the dependence on the input size n. Let's write insertion sort's running time as $c_1 n \cdot n$ and merge sort's running time as $c_2 n \cdot \lg n$. Then we see that where insertion sort has a factor of n in its running time, merge sort has a factor of $\lg n$, which is much smaller. (For example, when $n = 1000$, $\lg n$ is approximately 10, and when n equals one million, $\lg n$ is approximately only 20.) Although insertion sort usually runs faster than merge sort for small input sizes, once the input size n becomes large enough, merge sort's advantage of $\lg n$ vs. n will more than compensate for the difference in constant factors. No matter how much smaller c_1 is than c_2, there will always be a crossover point beyond which merge sort is faster.

Contoh (2)

For a concrete example, let us pit a faster computer (computer A) running insertion sort against a slower computer (computer B) running merge sort. They each must sort an array of 10 million numbers. (Although 10 million numbers might seem like a lot, if the numbers are eight-byte integers, then the input occupies about 80 megabytes, which fits in the memory of even an inexpensive laptop computer many times over.) Suppose that computer A executes 10 billion instructions per second (faster than any single sequential computer at the time of this writing) and computer B executes only 10 million instructions per second, so that computer A is 1000 times faster than computer B in raw computing power. To make the difference even more dramatic, suppose that the world's craftiest programmer codes insertion sort in machine language for computer A, and the resulting code requires $2n^2$ instructions to sort n numbers. Suppose further that just an average programmer implements merge sort, using a high-level language with an inefficient compiler, with the resulting code taking $50n \lg n$ instructions. To sort 10 million numbers, computer A takes

Contoh (3)

$$\frac{2 \cdot (10^7)^2 \text{ instructions}}{10^{10} \text{ instructions/second}} = 20{,}000 \text{ seconds (more than 5.5 hours)},$$

while computer B takes

$$\frac{50 \cdot 10^7 \lg 10^7 \text{ instructions}}{10^7 \text{ instructions/second}} \approx 1163 \text{ seconds (less than 20 minutes)}.$$

By using an algorithm whose running time grows more slowly, even with a poor compiler, computer B runs more than 17 times faster than computer A! The advantage of merge sort is even more pronounced when we sort 100 million numbers: where insertion sort takes more than 23 days, merge sort takes under four hours. In general, as the problem size increases, so does the relative advantage of merge sort.