Advanced Algorithms (I)

Yijia Chen
Fudan University
Yijia Chen

Homepage: http://basics.sjtu.edu.cn/~chen

Email: yijiachen@fudan.edu.cn
No text book.

Instead we will go through some papers on the forefront of algorithms research.
Edsger W. Dijkstra

*Computer Science is no more about computers than astronomy is about telescopes.*
Basics
Big-$O$ notation
Big-$O$ notation

$f(n)$ and $g(n)$ are the running times of two algorithms on inputs of size $n$.

Let $f : \mathbb{N}^+ \rightarrow \mathbb{R}^+$. We say $f = O(g)$ (which means that “$f$ grows no faster than $g$”) if there is a constant $c > 0$ such that $f(n) \leq c \cdot g(n)$.

$f = O(g)$ is a very loose analog of “$f \leq g$.” It differs from the usual notion of because of the constant $c$, so that for instance $10n = O(n)$. 
Why disregard the constant?

We are choosing between two algorithms with running time $f_1(n) = n^2$ and $f_2(n) = 2n + 20$ respectively.

Which is better?

$$f_1(n) \leq f_2(n) \text{ for } n \leq 5, \quad \text{and} \quad f_2(n) < f_1(n) \text{ for } n > 5.$$  

$f_2$ scales much better as $n$ grows, and therefore it is superior.

This superiority is captured by the big-$O$ notation: $f_2 = O(f_1)$:

$$\frac{f_2(n)}{f_1(n)} = \frac{2n + 20}{n^2} \leq 22$$

for all $n \in \mathbb{N}$. On the other hand, $f_1 \neq O(f_2)$, since the ratio

$$\frac{f_1(n)}{f_2(n)} = \frac{n^2}{2n + 20}$$

can get arbitrarily large.
Recall \( f_1(n) = n^2 \) and \( f_2(n) = 2n + 20 \). Suppose a third algorithm uses

\[
f_3(n) = n + 1
\]

steps. Is this better than \( f_2 \)? Certainly, but only by a constant factor.

The discrepancy between \( f_2 \) and \( f_3 \) is tiny compared to the huge gap between \( f_1 \) and \( f_2 \). In order to stay focused on the big picture, we treat functions as equivalent if they differ only by multiplicatively constants.

Returning to the definition of big-\( O \), we see that \( f_2 = O(f_3) \):

\[
\frac{f_2(n)}{f_3(n)} = \frac{2n + 20}{n + 1} \leq 20
\]

and \( f_3 = O(f_2) \) with \( c = 1 \).
Other similar notations

Just as $O(\cdot)$ is an analog of $\leq$, we also define analogs of $\geq$ and $=$ as follows:

- $f = \Omega(g)$ means $g = O(f)$,
- $f = \Theta(g)$ means $f = O(g)$ and $f = \Omega(g)$.

Recall $f_1(n) = n^2$, $f_2(n) = 2n + 20$ and $f_3(n) = n + 1$. Then

$$f_2 = \Theta(f_3) \quad \text{and} \quad f_1 = \Omega(f_3).$$
Divide and conquer
The divide-and-conquer strategy solves a problem by:

1. Breaking it into subproblems that are themselves smaller instances of the same type of problem.

2. Recursively solving these subproblems.

3. Appropriately combining their answers.
The algorithm

MERGESORT($a[1 \ldots n]$)
// Input: an array of numbers $a[1 \ldots n]$
// Output: A sorted version of this array
1. if $n > 1$ then
2. return MERGE(MERGESORT($a[1 \ldots \lfloor n/2 \rfloor]$),
3. MERGESORT($a[\lceil n/2 \rceil + 1 \ldots n]$)),
4. else return $a$.

MERGE($x[1 \ldots k], y[1 \ldots \ell]$)
// Input: two sorted arrays $x$ and $y$
// Output: A sorted version of the union of $x$ and $y$
1. if $k = 0$ then return $y[1 \ldots \ell]$
2. if $\ell = 0$ then return $x[1 \ldots k]$
3. if $x[1] \leq y[1]$
4. then return $x[1] \circ$MERGE($x[2 \ldots k], y[1 \ldots \ell]$)
5. else return $y[1] \circ$MERGE($x[1 \ldots k], y[2 \ldots \ell]$).
The time analysis

The recurrence relation:

\[ T(n) = 2T(n/2) + O(n); \]

By Master Theorem

\[ T(n) = O(n \log n). \]
Master theorem

If

\[ T(n) = aT(\lceil n/b \rceil) + O(n^d) \]

for some constants \( a > 0, \ b > 1, \) and \( d \geq 0, \) then

\[
T(n) = \begin{cases} 
O(n^d) & \text{if } d > \log_b a \\
O(n^d \log n) & \text{if } d = \log_b a \\
O(n^{\log_b a}) & \text{if } d < \log_b a.
\end{cases}
\]
Median
The median of a list of numbers is its 50th percentile: half the numbers are bigger than it, and half are smaller. In case the list has even length, there are two choices for what the middle element could be, in which case we pick the smaller of the two.

Computing the median of $n$ numbers by sorting takes $O(n \log n)$ time. Ideally we would like a linear time algorithm.
Selection

**Input:** A list of numbers $S$; an integer $k$.

**Output:** The $k$th smallest element of $S$. 
A randomized divide-and-conquer algorithm for selection

Any number $v$ splits the list $S$ into three categories:

- elements smaller than $v$, i.e., $S_L$;
- those equal to $v$, i.e., $S_v$ (there might be duplicates);
- and those greater than $v$, i.e., $S_R$ respectively.

\[
\text{selection}(S, k) = \begin{cases} 
\text{selection}(S_L, k) & \text{if } k \leq |S_L| \\
v & \text{if } |S_L| < k \leq |S_L| + |S_v| \\
\text{selection}(S_R, k - |S_L| - |S_v|) & \text{if } k > |S_L| + |S_v|.
\end{cases}
\]
How to choose \( \nu \)?

Ideally

\[ |S_L|, |S_R| \approx \frac{|S|}{2}. \]

And in that case

\[ T(n) = T(n/2) + O(n) = O(n). \]

But this requires picking \( \nu \) to be the median, which is our ultimate goal!

Instead, a much simpler alternative: pick \( \nu \) randomly from \( S \).
How to choose $v$? (cont’d)

Worst-case scenario

$$n + (n - 1) + (n - 2) + \cdots + 1 = \Theta(n^2)$$

Best-case scenario: $O(n)$.

Where, in this spectrum from $O(n)$ to $\Theta(n^2)$, does the average running time lie? Fortunately, it lies very close to the best-case time.
The efficiency analysis

\[ v \text{ is good if it lies within the 25th to 75th percentile of the array. A randomly chosen } v \text{ has a 50\% chance of being good}, \]

Lemma

*On average a fair coin needs to be tossed two times before a “heads” is seen.*

Proof.

\[ E := \text{expected number of tosses before head is seen.} \]

We need at least one toss, and it’s heads, we’re done. If it’s tail (with probability \(1/2\)), we need to repeat. Hence

\[ E = 1 + \frac{1}{2} E, \]

whose solution is \( E = 2 \)
The efficiency analysis (cont’d)

Let $T(n)$ be the expected running time on an array of size $n$, we get

$$T(n) \leq T(3n/4) + O(n) = O(n).$$
Depth-first search
The goal of depth-first search

What parts of the graph are reachable from a given vertex?
Exploring graphs

EXPLORE(G, v)
Input: G = (V, E) is a graph; v ∈ V
Output: visited(u) is set to TRUE for all nodes u reachable from v

1. visited(v) = TRUE
2. for each edge (v, u) ∈ E do
3. if not visited(u) then EXPLORE(u)
Theorem

EXPLORE\((G, v)\) is correct, i.e., it visits exactly all nodes that are reachable from \(v\).

Proof.

Every node which it visits must be reachable from \(v\):

EXPLORE only moves from nodes to their neighbors and can therefore never jump to a region that is not reachable from \(v\).

Every node which is reachable from \(v\) must be visited eventually:

If there is some \(u\) that EXPLORE misses, choose any path from \(v\) to \(u\), and look at the last vertex \(v\) on that path that the procedure visited. Let \(w\) be the node immediately after it on the same path.

So \(z\) was visited but \(w\) was not. This is a contradiction: while EXPLORE was at node \(z\), it would have noticed \(w\) and moved on to it. \(\square\)
Depth-first search

\[
\text{DFS}(G)
\]

1. for all \( v \in V \) do
2. \( \text{visited}(v) = \text{FALSE} \)
3. for all \( v \in V \) do
4. if not \( \text{visited}(v) \) then \( \text{EXPLORE}(v) \)
Running time of DFS

Because of the visited array, each vertex is EXPLORE’d just once.

During the exploration of a vertex, there are the following steps:

1. marking the spot as visited,

2. a loop in which adjacent edges are scanned.

The total work done in step 1 is then $O(|V|)$. In step 2, over the course of the entire DFS, each edge $\{x, y\} \in E$ is examined exactly twice, once during EXPLORE($x$) and once during EXPLORE($y$).

The overall time for step 2 is therefore $O(|E|)$ and so the depth-first search has a running time of $O(|V| + |E|)$. 
Breath-first search
DFS does not necessarily find the shortest paths.

Definition
The distance between two nodes is the length of the shortest path between them.
The algorithm

\textbf{BFS}(G, s)

\textbf{Input:} Graph $G = (V, E)$, directed or undirected; vertex $s \in V$

\textbf{Output:} For all vertices $u$ reachable from $s$, $\text{dist}(u)$ is set to the distance from $s$ to $u$.

1. \textbf{for all} $u \in V$ \textbf{do}
2. \hspace{1em} $\text{dist}(u) = \infty$
3. \hspace{1em} $\text{dist}(s) = 0$
4. $Q = [s]$ (queue containing just $s$)
5. \textbf{while} $Q$ is not empty \textbf{do}
6. \hspace{1em} $u = \text{eject}(Q)$
7. \hspace{1em} \textbf{for all} edge $(u, v) \in E$ \textbf{do}
8. \hspace{2em} \textbf{if} $\text{dist}(v) = \infty$ \textbf{then}
9. \hspace{3em} $\text{inject}(Q, v)$
10. \hspace{2em} $\text{dist}(v) = \text{dist}(u) + 1$
Correctness and efficiency

Lemma
For each $d = 0, 1, 2, \ldots$, there is a moment at which (1) all nodes at distance $\leq d$ from $s$ have their distances correctly set; (2) all other nodes have their distances set to $\infty$; and (3) the queue contains exactly the nodes at distance $d$.

Lemma
BFS has a running time of $O(|V| + |E|)$. 
Dynamic programming
Longest increasing subsequences
In the longest increasing subsequence problem, the input is a sequence of numbers \( a_1, \ldots, a_n \).

A subsequence is any subset of these numbers taken in order, of the form

\[ a_{i_1}, a_{i_2}, \ldots, a_{i_k} \]

where \( 1 \leq i_1 < i_2 < \cdots < i_k \leq n \), and an increasing subsequence is one in which the numbers are getting strictly larger.

The task is to find the increasing subsequence of greatest length.
Graph reformulation

Create a graph of all permissible transitions: establish a node $i$ for each element $a_i$, and add directed edges $(i, j)$ whenever it is possible for $a_i$ and $a_j$ to be consecutive elements in an increasing subsequence:

$$i < j \text{ and } a_i < a_j$$

- This graph $G = (V, E)$ is a directed acyclic graph (dag), since all edges $(i, j)$ have $i < j$
- There is a one-to-one correspondence between increasing subsequences and paths in this dag.

Therefore, our goal is simply to find the longest path in the dag!
for $j = 1$ to $n$ do
   $L(j) = 1 + \max \{ L(i) \mid (i, j) \in E \}$
return $\max_j L(j)$

$L(j)$ is the length of the longest path – the longest increasing subsequence – ending at $j$ plus 1.

Any path to node $j$ must pass through one of its predecessors, and therefore $L(j)$ is $1$ plus the maximum $L(\cdot)$ value of these predecessors. If there are no edges into $j$, take the maximum over the empty set, i.e., zero.

The final answer is the largest $L(j)$, since any ending position is allowed.
In order to solve our original problem, we have defined a collection of subproblems \( \{L(j) \mid 1 \leq j \leq n\} \) with the following key property that allows them to be solved in a single pass:

*There is an ordering on the subproblems, and a relation that shows how to solve a subproblem given the answers to “smaller” subproblems, that is, subproblems that appear earlier in the ordering.*

In our case, each subproblem is solved using the relation

\[
L(j) = 1 + \max\{L(i) \mid (i, j) \in E\}
\]
Independent sets in trees
It is well known that many NP-hard problems can be solved in polynomial time on trees, i.e., Independent-Set, Dominating-Set, 3-Colorability, etc.
Independent sets in trees

Recall that a subset $S \subseteq V$ is an independent set of graph $G = (V, E)$ if there are no edges between vertices in $S$.

Simple dynamic programming on trees:

Given a tree $T$ we first fix an arbitrary $r \in V(T)$ as the root. Then we compute from leaves to the root $r$ for each node $t$

$$I(t) := \text{size of a largest independent set of the subtree hanging from } t$$

$$= \max \left\{ 1 + \sum_{\text{grandchildren } t' \text{ of } t} I(t'), \sum_{\text{children } t' \text{ of } t} I(t') \right\}.$$
Graph Isomorphism
Graph isomorphism

Definition
Let $G$ and $H$ be two graphs. A function $f : V(G) \to V(H)$ is an isomorphism if

(GI1) $f$ is a bijection;

(GI2) for every $u, v \in V(G)$ we have $\{u, v\} \in E(G)$ if and only if $\{f(u), f(v)\} \in E(H)$.

If such an $f$ exists, then $G$ and $H$ are isomorphic.
Graph Isomorphism (GI) problem

GI

Input: Two graphs $G$ and $H$.

Problem: Decides whether $G$ and $H$ are isomorphic.

Remark

1. GI is in NP.
2. GI is not NP-complete, unless Polynomial Hierarchy collapses (which most people do not believe).
3. We don’t know whether GI is P-hard.
4. [Babai, 2015] GI can be decided in time $n^{\log^{O(1)} n}$ on two $n$-vertex graphs.
Color refinement
The algorithm

1. Initially, all vertices have the same color.

2. Then in each step of the iteration, two vertices that currently have the same color get different colors if for some color $c$ they have a different number of neighbours of color $c$.

3. The process stops if no further refinement is achieved, resulting in a stable coloring of the graph.
GI by color refinement

1. Run color refinement on the disjoint union of two graphs $G$ and $H$.

2. If the stable coloring differs on $G$ and $H$, that is, if for some color $c$, the graphs have a different number of vertices of color $c$, then they are nonisomorphic.