Ubuntu Algorithm Classes

Notes
25.05.2012

Classes will be on 25th of May (Friday)
Server: IRC freenode, #ubuntu-classroom (http://webchat.freenode.net)
The survey link is http://www.surveymonkey.com/s/QC25SCB
Information: http://bdfhjk.blog.pl

1. Exponentiation by squaring

Wiki: http://en.wikipedia.org/wiki/Exponentiation_by_squaring
Exponentiation by squaring is an algorithm. It's used for fast computation of large integer powers of a number. It's also known as "binary exponentiation".

For example, we want to compute 5^{10}. A naive method is to multiply 5 ten times, so we just need to compute $5 \cdot 5 \cdot 5 \cdot 5 \cdot 5 \cdot 5 \cdot 5 \cdot 5 \cdot 5 \cdot 5$. If you want to compute x^n using this method, you need O(n) operations.

We can notice that O(n) operations when computing x^n is slow when n is really big, for example $n > 10^6$. But, if n is even, we can notice that:

$$x^n = \left(x^{\frac{n}{2}}\right)^2$$

This gives us a possibility to calculate x^n much faster than with O(n) operations.
We can also notice that is n is odd, this is true:

$$x^n = x \cdot \left(x^{\frac{n-1}{2}}\right)^2$$

If $n = 0, x^n = 1$.

So, if we run it recursively, we can calculate x^n quite fast. This is the C++ code of recursive function which counts the amount od x^n:

```
long long power(long long x, long long n)
{
	if(n == 0)
		return 1;
	if(n % 2) // n is odd
		return (x * power(x, n - 1)) % Q;
	else // n is even
	{
		int a = power(x, n / 2);
		return (a * a) % Q;
	}
}
```

How does it work? I'll explain it using an example of computing 5^{10}:

```
power(5, 10)
x = 5
n = 10
n is even
a = power(5, 5)
	n = 5
	n is odd
	return x * power(5, 4)
		n = 4
		n is even
		a = power(5, 2)
			n = 2
			n is even
			a = power(5, 1)
				n = 1
				n is odd
				return x * power(5, 0)
					n = 0
					power(5, 0) = 1;
				power(5, 1) = 5 * 1 = 5
```

a = 5
power(5, 2) = 5 * 5 = 25
a = 25
power(5, 4) = 25 * 25 = 625
power(5, 5) = 5 * 625 = 3125
a = 3125
power(5, 10) = 3125 * 3125 = 9765625

First we calculate $5^1 = 5$, then $5^2 = 5^1 \cdot 5^1 = 25$, $5^4 = 5^2 \cdot 5^2 = 625$, $5^5 = 5^4 \cdot 5 = 3125$, $5^{10} = 5^5 \cdot 5^5 = 9765625$.

This gives us 4 multiplications, not 9 as in naive algorithm. Complexity of exponentiation by squaring is O($log_2 n$). For example, if we want to compute $x^{1000000}$, we need 999999 multiplications using naive algorithm, but using binary exponentiation we need only about 20 multiplications.

There is also an iterative version of this algorithm, but it's not that clear and it's harder to understand, so I won't explain how does it work. The recursive version is easier, but it's slower and it uses more memory. Here is C++ code of iterative version:

```
int power(int x, int n)
{
	int a = 1;
	while(n > 0)
	{
		if(n % 2)
			a *= x;
		x *= x;
		n /= 2;
	}
	return a;
}
```

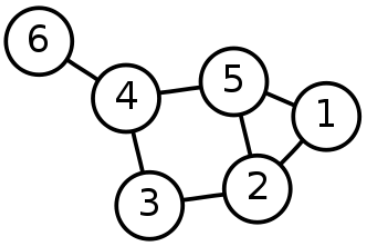
2. Adjacency list

Wiki: http://en.wikipedia.org/wiki/Adjacency_list

During latest classes, there was something about graphs and how to represent a graph. The simpliest way to do it is an adjacency matrix, but it was discussed during the last classes.

Adjacency matrix is surely the simpliest way to represent a graph in computer's memory. But using it isn't very fast and it uses much memory. The amount of used memory is O(n^2) for adjacency matrix and O(n + m) for adjacency list, what makes a big difference.

How does it work? This is an example of simple, undirected graph:



Adjacency list for this graph will look like this:

1	adjacent to	2	5	
2	adjacent to	1	3	5
3	adjacent to	2	4	
4	adjacent to	3	5	6
5	adjacent to	1	2	4
6	adjacent to	4		

How to keep it in computer's memory? It's quite simple. We need to make *n* lists (n is a number of vertices in our graph) and list *E[i]* will contain all nodes adjacent to vertex *i*.
In C++, it looks like this (works for undirected graphs):

```
int main()
{
	int n, m; // n - number of nodes, m - number of edges
	scanf("%d%d", &n, &m);
	std::vector<int> E[n + 1];
	for(int i = 0; i < m; i++)
	{
		int u, v;
		scanf("%d%d", &u, &v);
		E[u].push_back(v);
		E[v].push_back(u); // delete this line for directed graph
	}
	return 0;
}
```