Diophantine – not NP-Complete (unsolvable)

$Diophantine = \{p \mid p \text{ is a polynomial with an integral root}\}$
The mapping reduction 3-SAT $\leq_P$ *Diophantine* is achieved as follows.
Consider a 3-CNF formula φ:

- Each literal of φ, x, maps to integer variable x.
- Each literal $\overline{x}$ maps to expression $(1 - x)$.
- Each $\vee$ in a clause maps to . (times).
- Each clause is mapped as above, and then *squared.*
- Each $\wedge$ maps to $+$.
- The resulting equation is set to 0.
- Example: map

$$\varphi = (x \vee y) \wedge (x \vee \overline{y}) \wedge (\overline{x} \vee \overline{y})$$

to

$$E = (x.y)^2 + (x.(1-y))^2 + ((1-x).(1-y))^2 = 0.$$

Chapter 4 recurrences – substitution method

1. Guess a solution [this part requires some experiences, exercises, etc]
2. Prove the solution is correct

case $T(n) = b$ *for* $n < 2$*):*

$$T(n) = 2T(n/2) + bn\log n.$$

This looks very similar to the recurrence relation for the merge-sort routine, so we might make as our first guess the following:

$$\textit{First guess: } T(n) \leq cn\log n,$$

case $T(n) = b$ for $n < 2$):

$$T(n) = 2T(n/2) + bn\log n.$$

This looks very similar to the recurrence relation for the merge-sort routine, so we might make as our first guess the following:

First guess: $T(n) \leq cn\log n,$

$$\begin{aligned} T(n) &= 2T(n/2) + bn\log n \\ &\leq 2(c(n/2)\log(n/2)) + bn\log n \\ &= cn(\log n - \log 2) + bn\log n \\ &= cn\log n - cn + bn\log n. \end{aligned}$$

not working!

case $T(n) = b$ for $n < 2$):

$$T(n) = 2T(n/2) + bn\log n.$$

This looks very similar to the recurrence relation for the merge-sort routine, so we might make as our first guess the following:

$$\textit{First guess: } T(n) \leq cn\log n,$$

$$\textit{Better guess: } T(n) \leq cn\log^2 n,$$

$$\begin{aligned} T(n) &= 2T(n/2) + bn\log n \\ &\leq 2(c(n/2)\log^2(n/2)) + bn\log n \\ &= cn(\log^2 n - 2\log n + 1) + bn\log n \\ &= cn\log^2 n - 2cn\log n + cn + bn\log n \\ &\leq cn\log^2 n, \end{aligned}$$

provided $c \geq b$. Thus, we have shown that $T(n)$ is indeed $O(n\log^2 n)$.

Worst time complexity of Quicksort – section 7.4

```
QUICKSORT(A, p, r)
1  if p < r
2     then q ← PARTITION(A, p, r)
3          QUICKSORT(A, p, q − 1)
4          QUICKSORT(A, q + 1, r)
```

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4       do if A[j] ≤ x
5            then i ← i + 1
6                 exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return i + 1
```

time of quicksort is $O(n^2)$. Let $T(n)$ be the worst-case time for the procedure QUICKSORT on an input of size n. We have the recurrence

$$T(n) = \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + \Theta(n) , \tag{7.1}$$

where the parameter q ranges from 0 to $n-1$ because the procedure PARTITION produces two subproblems with total size $n-1$. We guess that $T(n) \le cn^2$ for some constant c. Substituting this guess into recurrence (7.1), we obtain

$$\begin{aligned} T(n) &\le \max_{0 \le q \le n-1} (cq^2 + c(n-q-1)^2) + \Theta(n) \\ &= c \cdot \max_{0 \le q \le n-1} (q^2 + (n-q-1)^2) + \Theta(n) . \end{aligned}$$

The expression $q^2 + (n-q-1)^2$ achieves a maximum over the parameter's range $0 \le q \le n-1$ at either endpoint, as can be seen since the second derivative of the expression with respect to q is positive (see Exercise 7.4-3). This observation gives us the bound $\max_{0 \le q \le n-1}(q^2 + (n-q-1)^2) \le (n-1)^2 = n^2 - 2n + 1$. Continuing with our bounding of $T(n)$, we obtain

$$\begin{aligned} T(n) &\le cn^2 - c(2n-1) + \Theta(n) \\ &\le cn^2 , \end{aligned}$$

master method – section 4.3

Theorem 4.1 (Master theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ can be bounded asymptotically as follows.

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

$$T(n) = 9T(n/3) + n\ .$$

For this recurrence, we have $a = 9$, $b = 3$, $f(n) = n$, and thus we have that $n^{\log_b a} = n^{\log_3 9} = \Theta(n^2)$. Since $f(n) = O(n^{\log_3 9 - \epsilon})$, where $\epsilon = 1$, we can apply case 1 of the master theorem and conclude that the solution is $T(n) = \Theta(n^2)$.

Now consider

$$T(n) = T(2n/3) + 1,$$

in which $a = 1$, $b = 3/2$, $f(n) = 1$, and $n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$. Case 2 applies, since $f(n) = \Theta(n^{\log_b a}) = \Theta(1)$, and thus the solution to the recurrence is $T(n) = \Theta(\lg n)$.

$$T(n) = 3T(n/4) + n \lg n ,$$

we have $a = 3$, $b = 4$, $f(n) = n \lg n$, and $n^{\log_b a} = n^{\log_4 3} = O(n^{0.793})$. Since $f(n) = \Omega(n^{\log_4 3+\epsilon})$, where $\epsilon \approx 0.2$, case 3 applies if we can show that the regularity condition holds for $f(n)$. For sufficiently large n, $af(n/b) = 3(n/4) \lg(n/4) \leq (3/4)n \lg n = cf(n)$ for $c = 3/4$. Consequently, by case 3, the solution to the recurrence is $T(n) = \Theta(n \lg n)$.

The master method does not apply to the recurrence

$$T(n) = 2T(n/2) + n \lg n ,$$

even though it has the proper form: $a = 2$, $b = 2$, $f(n) = n \lg n$, and $n^{\log_b a} = n$. It might seem that case 3 should apply, since $f(n) = n \lg n$ is asymptotically larger than $n^{\log_b a} = n$. The problem is that it is not *polynomially* larger. The ratio

chapter 11 hash table – an example of a dictionary

Many applications require a dynamic set that supports only the dictionary operations INSERT, SEARCH, and DELETE. For example, a compiler for a computer language maintains a symbol table, in which the keys of elements are arbitrary character strings that correspond to identifiers in the language. A hash table is an effective data structure for implementing dictionaries. Although searching for an element in a hash table can take as long as searching for an element in a linked list—$\Theta(n)$ time in the worst case—in practice, hashing performs extremely well. Under reasonable assumptions, the expected time to search for an element in a hash table is $O(1)$.

To represent the dynamic set, we use an array, or ***direct-address table***, denoted by $T[0..m-1]$, in which each position, or ***slot***, corresponds to a key in the universe U. Figure 11.1 illustrates the approach; slot k points to an element in the set with key k. If the set contains no element with key k, then $T[k] = \text{NIL}$.

The dictionary operations are trivial to implement.

```
DIRECT-ADDRESS-SEARCH(T, k)
    return T[k]

DIRECT-ADDRESS-INSERT(T, x)
    T[key[x]] ← x

DIRECT-ADDRESS-DELETE(T, x)
    T[key[x]] ← NIL
```

Each of these operations is fast: only $O(1)$ time is required.

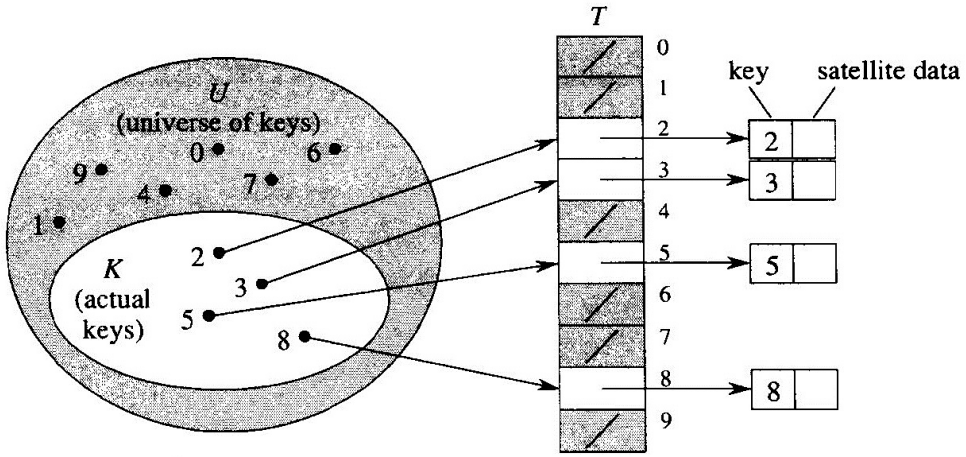


Figure 11.1 Implementing a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded, contain NIL.

Direct address table – space / time trade-off

Time – O(1) : very fast (worst case complexity)
Space – impractical

 a hash table

Time – O(1) : expected time
Space – “more” efficient than direct address table

h : U -> {0,1,…,m-1}

Collision – two different keys may hash to the same slot

The difficulty with direct addressing is obvious: if the universe U is large, storing a table T of size $|U|$ may be impractical, or even impossible, given the memory available on a typical computer. Furthermore, the set K of keys *actually stored* may be so small relative to U that most of the space allocated for T would be wasted.

When the set K of keys stored in a dictionary is much smaller than the universe U of all possible keys, a hash table requires much less storage than a direct-address table. Specifically, the storage requirements can be reduced to $\Theta(|K|)$ while we maintain the benefit that searching for an element in the hash table still requires only $O(1)$ time. The only catch is that this bound is for the *average time*, whereas for direct addressing it holds for the *worst-case time*.

With direct addressing, an element with key k is stored in slot k. With hashing, this element is stored in slot $h(k)$; that is, we use a ***hash function*** h to compute the slot from the key k. Here h maps the universe U of keys into the slots of a ***hash table*** $T[0..m-1]$:

$$h : U \to \{0, 1, \ldots, m-1\} .$$

We say that an element with key k ***hashes*** to slot $h(k)$; we also say that $h(k)$ is the ***hash value*** of key k. Figure 11.2 illustrates the basic idea. The point of

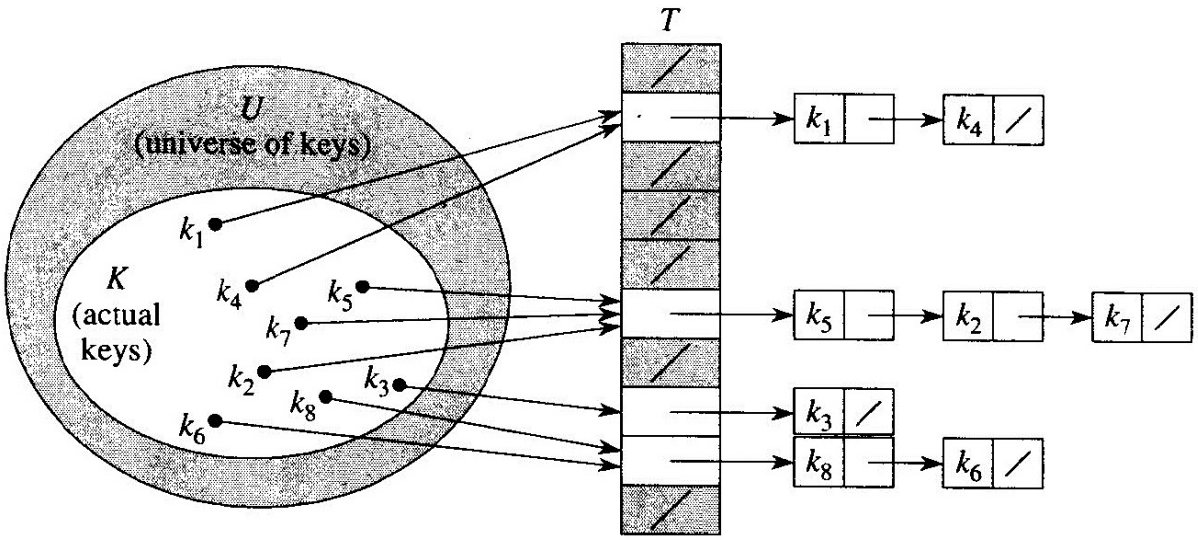


Figure 11.3 Collision resolution by chaining. Each hash-table slot $T[j]$ contains a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_2) = h(k_7)$.

Collision resolution methods

(1) Chaining – maintain a linked list – insertion / deletion O(1) time, worst complexity

CHAINED-HASH-INSERT(T, x)
insert x at the head of list $T[h(key[x])]$

CHAINED-HASH-SEARCH(T, k)
search for an element with key k in list $T[h(k)]$

CHAINED-HASH-DELETE(T, x)
delete x from the list $T[h(key[x])]$

(2) Open addressing – find a next available slot (computing a probe sequence)

Search – chaining [expected time – O(1) time]

Theorem 11.1

In a hash table in which collisions are resolved by chaining, an unsuccessful search takes expected time $\Theta(1 + \alpha)$, under the assumption of simple uniform hashing.

Theorem 11.2

In a hash table in which collisions are resolved by chaining, a successful search takes time $\Theta(1 + \alpha)$, on the average, under the assumption of simple uniform hashing.

load factor = (number of keys) / (number of slots)

simple uniform hashing – each key is equally likely to be hashed to any one of m slots AND for any two keys a and b, where a hashes to is independent of where b hashes to

11.3 hash function

division method

$h(k) = k \bmod m$

m should not be a power of 2.

Ex. 1100 = 12
10100 = 20
11100 = 28
101100 = ~~48~~ 44

$h(k) = k \bmod 8$

12 mod 8 = 4
20 mod 8 = 4
28 mod 8 = 4
~~48~~ 44 mod 8 = 04

multiplication method

$$h(k) = \lfloor m\,(kA \bmod 1) \rfloor$$

$$kA \bmod 1 = kA - \lfloor kA \rfloor$$

Ex. $k = 12$, $A = 0.6$

$0 < A < 1$

$7.2 - 7 = 0.2$

11.4 open addressing

In ***open addressing***, all elements are stored in the hash table itself. That is, each table entry contains either an element of the dynamic set or NIL. When searching for an element, we systematically examine table slots until the desired element is found or it is clear that the element is not in the table. There are no lists and no elements stored outside the table, as there are in chaining. Thus, in open addressing, the hash table can "fill up" so that no further insertions can be made; the load factor α can never exceed 1.

hash function is extended

$h: U \times \{0, 1, 2, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$

$h(\uparrow, \uparrow)$

key, probe #

$(h\langle k, 0\rangle, h\langle k, 1\rangle, \ldots, h\langle k, m-1\rangle)$

~ probe sequence

insert / search – O(n)

```
HASH-INSERT(T, k)
1  i ← 0
2  repeat  j ← h(k, i)
3          if T[j] = NIL
4            then T[j] ← k
5                 return j
6            else  i ← i + 1
7    until i = m
8  error "hash table overflow"
```

```
HASH-SEARCH(T, k)
1  i ← 0
2  repeat  j ← h(k, i)
3          if T[j] = k
4            then return j
5          i ← i + 1
6    until T[j] = NIL or i = m
7  return NIL
```

deletion should be done carefully

3 ways to compute probe sequences

In our analysis, we make the assumption of ***uniform hashing***: we assume that each key is equally likely to have any of the $m!$ permutations of $\langle 0, 1, \ldots, m-1 \rangle$ as its probe sequence. Uniform hashing generalizes the notion of simple uniform

linear probing, quadratic probing, double hashing

Given an ordinary hash function $h' : U \to \{0, 1, \ldots, m-1\}$, which we refer to as an ***auxiliary hash function***, the method of ***linear probing*** uses the hash function

$$h(k, i) = (h'(k) + i) \bmod m$$

for $i = 0, 1, \ldots, m-1$. Given key k, the first slot probed is $T[h'(k)]$, i.e., the

Quadratic probing uses a hash function of the form

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m , \tag{11.5}$$

where h' is an auxiliary hash function, c_1 and $c_2 \neq 0$ are auxiliary constants, and $i = 0, 1, \ldots, m - 1$. The initial position probed is $T[h'(k)]$; later positions probed

Double hashing is one of the best methods available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations. ***Double hashing*** uses a hash function of the form

$$h(k, i) = (h_1(k) + i h_2(k)) \bmod m ,$$

where h_1 and h_2 are auxiliary hash functions. The initial probe is to position

The value $h_2(k)$ must be relatively prime to the hash-table size m for the entire hash table to be searched. (See Exercise 11.4-3.) A convenient way to ensure this

search – expected time [very fast!]

Theorem 11.6

Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.

Theorem 11.8

Given an open-address hash table with load factor $\alpha < 1$, the expected number of probes in a successful search is at most

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha},$$

assuming uniform hashing and assuming that each key in the table is equally likely to be searched for.

Quantum computer – a computer which is governed by
quantum mechanical properties (superposition, entanglement, etc)

Basic unit of information – qubit (quantum bit)
0, 1, superposition of both

Classical computer – a computer which is governed by classical physics (
Newton's law, etc)

Basic unit of information – bit (either 1 or 0)

Implicit assumptions in the theory of classical computation.

Assumption	Classically	Quantum Mechanically
A bit always has a definite value.	True	False. A bit need not have a definite value until the moment after it is observed.
A bit can only be 0 or 1.	True	False. A bit can be in a superposition of 0 and 1 simultaneously.
A bit can be copied without affecting its value.	True	False. A qubit in an unknown state cannot be copied without necessarily disrupting its state.
A bit can be read without affecting its value.	True	False. Reading a qubit that is initially in a superposition will change the value of the qubit.
Reading one bit of the computer has no affect on another (unread) bit.	True	False. If the bit being read is entangled with another qubit, reading one qubit will affect the other.

Principle of superposition

qubit it is always found to be in one of two states: 0 or 1. The Principle of Superposition says that if a quantum system can be measured to be in one of a number of states, then it can also exist in a blend, or superposition, of all of its observable states simultaneously. In the context of quantum computing, superposition means that an n-qubit memory register can exist in a superposition of all 2^n possible configurations of n classical bits. Superposition therefore allows a quantum computer to operate like a massively parallel computer, working on several (classical) bit string inputs at once. Unfortunately, it is impossible to observe these parallel computations individually, so we are severely limited in the information we can extract from the parallel computations.

entanglement

The third phenomenon is *entanglement.* If two or more qubits are made to interact, they can emerge from the interaction in a definite *joint* quantum state that cannot be expressed in terms of a product of definite *individual* quantum states. Thus, the state of the component qubits can be fuzzy, even though the state of the collection of qubits as a whole is well defined. A consequence of entanglement is that a pair of entangled qubits retain a lingering, instantaneous influence on one another no matter how far apart they become and regardless of the nature of the intervening medium. If one of the qubits is measured, for example, then *at that instant,* the state of the qubit with which it is entangled becomes definite and, consequently, completely predictable, even though the two qubits might be in different galaxies. Such ghostly "non-local" effects, as they are called, are the

Quantum register - memory

To build a quantum computer, we use bits to represent data in the same way as in the classical computer. We also work with registers that are called **quantum registers** here. A quantum register can store one **quantum bit**. To distinguish quantum bits from classical ones, 0 and 1, one uses the notation

$|0\rangle$ and $|1\rangle$.

$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ $|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$

$\langle 0| = (1 \ 0)$ $\langle 1| = (0 \ 1)$

Ket notation / Bra notation

|0> ket vector
<0| bra vector

Bracket

One says that a quantum bit (as the contents of a quantum register) can be a **superposition** (or in a combination) of two classical bits $|0\rangle$ and $|1\rangle$. We describe superposition by

$$\alpha \cdot |0\rangle \quad + \quad \beta \cdot |1\rangle \quad ,$$

where α and β are complex numbers satisfying the properties

$$|\alpha|^2 \leq 1, |\beta|^2 \leq 1 \text{ and } |\alpha|^2 + |\beta|^2 = 1 \ .$$

$$\alpha^2 \leq 1, \beta^2 \leq 1 \text{ and } \alpha^2 + \beta^2 = 1 \ .$$

The values α and β are called **amplitudes**, and they estimate to which extent the quantum bit is $|0\rangle$ and to which extent it is $|1\rangle$.

Example 9.1 The superposition

$$\frac{1}{\sqrt{2}} \cdot |0\rangle + \frac{1}{\sqrt{2}} \cdot |1\rangle$$

expresses the fact that the quantum bit has the same probability of being in the classical state $|0\rangle$ as in $|1\rangle$.

$$\alpha^2 = \left(\frac{1}{\sqrt{2}}\right)^2 = \frac{1}{(\sqrt{2})^2} = \frac{1}{2} \quad \text{and} \quad \beta^2 = \left(\frac{1}{\sqrt{2}}\right)^2 = \frac{1}{2} \quad .$$

Example 9.2 The superposition

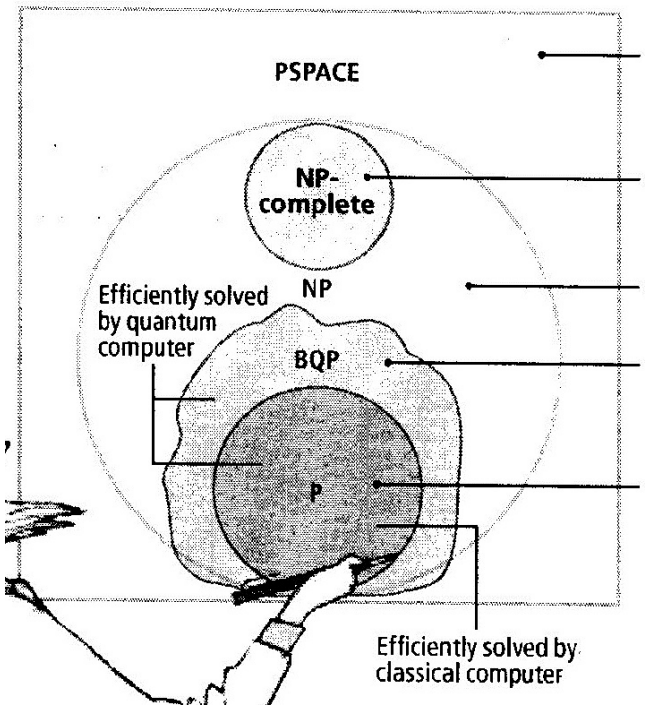
$$\frac{1}{\sqrt{3}} \cdot |0\rangle + \sqrt{\frac{2}{3}} \cdot |1\rangle$$

expresses the fact that after a measurement, one sees the outcome $|0\rangle$ with the probability

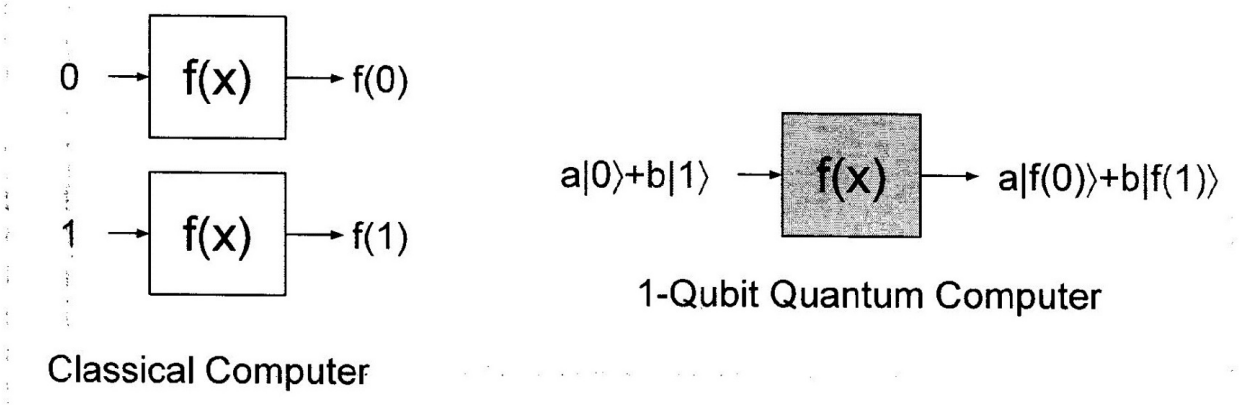
$$\alpha^2 = \left(\frac{1}{\sqrt{3}}\right)^2 = \frac{1}{3}$$

and $|1\rangle$ with the probability

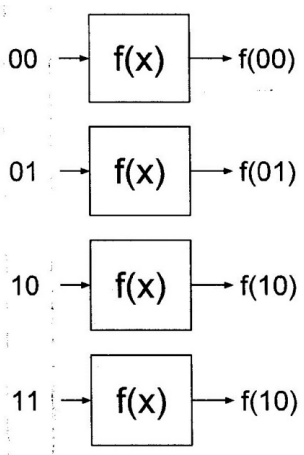
$$\beta^2 = \left(\sqrt{\frac{2}{3}}\right)^2 = \frac{2}{3} \, .$$


PSPACE
NP-
complete
NP
Efficiently solved
by quantum
computer
BQP
P
Efficiently solved by
classical computer

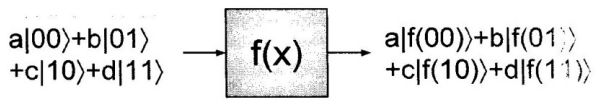
Difference between a classical computer and a quantum computer



Difference between a classical computer and a quantum computer



Classical Computer

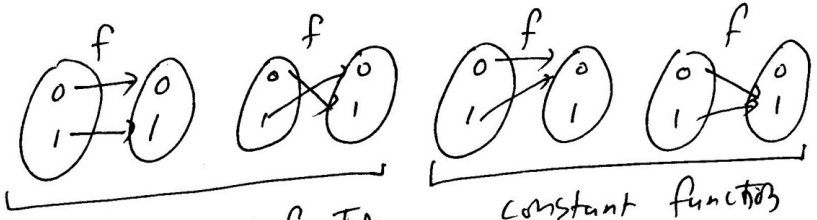


2-Qubit Quantum Computer

Deutsch's problem

Given $f: \{0,1\} \to \{0,1\}$

Is f a constant function
or a balanced function?



4 possibilities

On a classical computer,
$f(0)$, $f(1)$ need to be computed
2 computations

On a quantum computer
1 computation

How?

Deutsch's algorithm

1. Preparation

   $|\psi_1\rangle = |0\rangle|1\rangle$ 2 qubits

2. Use Hadamard transformations defined

   $H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$ for each qubit

$$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

$$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}\begin{pmatrix} 0 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

$$\frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix} = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$$

$$\frac{1}{\sqrt{2}}\begin{pmatrix}1\\-1\end{pmatrix} = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

$$|\psi_2\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

tensor product

tensor product operation

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1s} \\ a_{21} & a_{22} & \dots & a_{2s} \\ . & . & . & . \\ . & . & . & . \\ a_{r1} & a_{r2} & \dots & a_{rs} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1u} \\ b_{21} & b_{22} & \dots & b_{2u} \\ . & . & . & . \\ . & . & . & . \\ b_{t1} & b_{t2} & \dots & b_{tu} \end{pmatrix}$$

$$A \otimes B = \begin{pmatrix} a_{11}B & a_{12}B & \dots & a_{1s}B \\ a_{21}B & a_{22}B & \dots & a_{2s}B \\ . & . & \dots & \\ . & . & \dots & \\ a_{r1}B & a_{r2}B & \dots & a_{rs}B \end{pmatrix}$$

Ex $A=\begin{pmatrix}1&2&1\\2&2&1\end{pmatrix}$ $B=\begin{pmatrix}1&2\\3&4\end{pmatrix}$

$$A\otimes B=\begin{pmatrix}1\begin{pmatrix}1&2\\3&4\end{pmatrix} & 2\begin{pmatrix}1&2\\3&4\end{pmatrix} & 1\begin{pmatrix}1&2\\3&4\end{pmatrix}\\ 2\begin{pmatrix}1&2\\3&4\end{pmatrix} & 2\begin{pmatrix}1&2\\3&4\end{pmatrix} & 1\begin{pmatrix}1&2\\3&4\end{pmatrix}\end{pmatrix}$$

$$=\begin{pmatrix}1&2&2&4&1&2\\3&4&6&8&3&4\\2&4&2&4&1&2\\6&8&6&8&3&4\end{pmatrix}$$

$$|\psi_2\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

Use U_f defined as follows:

$$U_f : |x\rangle|y\rangle = |x\rangle|y \oplus f(x)\rangle$$

Exclusive OR

$$0 \oplus 0 = 0 \quad 0 \oplus 1 = 1$$
$$1 \oplus 0 = 1 \quad 1 \oplus 1 = 0$$

$$|\psi_3\rangle = U_f\left(\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\right)$$

$$= U_f\left(\frac{1}{2}(|00\rangle + |10\rangle - |01\rangle - |11\rangle)\right)$$

$$= \frac{1}{2}(U_f|00\rangle + U_f|10\rangle - U_f|01\rangle - U_f|11\rangle)$$

$U_f : |x\rangle|y\rangle = |x\rangle|y \oplus f(x)\rangle$

$U_f|00\rangle = |0\rangle|0 \oplus f(0)\rangle$

$U_f|10\rangle = |1\rangle|0 \oplus f(1)\rangle$

$U_f|01\rangle = |0\rangle|1 \oplus f(0)\rangle$

$U_f|11\rangle = |1\rangle|1 \oplus f(1)\rangle$

① $f(0)=f(1)=0$ $U_f|00\rangle = |0\rangle|0\rangle$

$U_f|10\rangle = |1\rangle|0\rangle$

$U_f|01\rangle = |0\rangle|1\rangle$

$U_f|11\rangle = |1\rangle|1\rangle$

② $f(0)=f(1)=1$ $U_f|00\rangle = |0\rangle|1\rangle$

$U_f|10\rangle = |1\rangle|1\rangle$

$U_f|01\rangle = |0\rangle|0\rangle$

$U_f|11\rangle = |1\rangle|0\rangle$

③ $f(0)=0, f(1)=1$ $U_f|00\rangle = |0\rangle|0\rangle$

$U_f|10\rangle = |1\rangle|1\rangle$

$U_f|01\rangle = |0\rangle|1\rangle$

$U_f|11\rangle = |1\rangle|0\rangle$

④ $f(0)=1, f(1)=0$ $U_f|00\rangle = |0\rangle|1\rangle$

$U_f|10\rangle = |1\rangle|0\rangle$

$U_f|01\rangle = |0\rangle|0\rangle$

$U_f|11\rangle = |1\rangle|1\rangle$

cases ①,② constant function

$$|\psi_3\rangle = \frac{1}{2}\left(U_f|00\rangle + U_f|10\rangle - U_f|01\rangle - U_f|11\rangle\right)$$

$$\frac{1}{2}\left(|00\rangle + |10\rangle - |01\rangle - |11\rangle\right)$$

or

$$\frac{1}{2}\left(|01\rangle + |11\rangle - |00\rangle - |10\rangle\right)$$

cases ③④ balanced function

$$\frac{1}{2}(|00\rangle + |11\rangle - |01\rangle - |10\rangle)$$

or

$$\frac{1}{2}(|01\rangle + |10\rangle - |00\rangle - |11\rangle)$$

$$|\psi_3\rangle = U_f |\psi_2\rangle$$

$$= U_f\left(\left(\frac{1}{\sqrt{2}}|0\rangle + |1\rangle\right) \otimes \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\right)$$

$$= U_f\left(\frac{1}{2}|00\rangle + |10\rangle - |01\rangle - |11\rangle\right)$$

therefore if f is a constant function

$$|\psi_3\rangle = \pm\frac{1}{\sqrt{2}}(|0\rangle+|1\rangle)\otimes\frac{1}{\sqrt{2}}(|0\rangle-|1\rangle)$$

if f is a balanced function

$$|\psi_3\rangle = \pm\frac{1}{\sqrt{2}}(|0\rangle-|1\rangle)\otimes\frac{1}{\sqrt{2}}(|0\rangle-|1\rangle)$$

$$\therefore |\psi_3\rangle = \frac{1}{\sqrt{2}}\left((-1)^{f(0)}|0\rangle+(-1)^{f(1)}|1\rangle\right)\otimes\frac{1}{\sqrt{2}}(|0\rangle-|1\rangle)$$

Hadamard transformation H is applied to the 1st qubit of $|\psi_3\rangle$

$$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \frac{1}{\sqrt{2}}\left((-1)^{f(0)}|0\rangle + (-1)^{f(1)}|1\rangle\right)$$

$$\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}\frac{1}{\sqrt{2}}\left((-1)^{f(0)}\begin{pmatrix}1\\0\end{pmatrix}+(-1)^{f(1)}\begin{pmatrix}0\\1\end{pmatrix}\right)$$

$$=\frac{1}{\sqrt{2}}\left((-1)^{f(0)}\begin{pmatrix}\frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}}\end{pmatrix}+(-1)^{f(1)}\begin{pmatrix}\frac{1}{\sqrt{2}}\\ -\frac{1}{\sqrt{2}}\end{pmatrix}\right)$$

Therefore

$$|\psi_f\rangle=\frac{1}{\sqrt{2}}\left((-1)^{f(0)}\begin{pmatrix}\frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}}\end{pmatrix}+(-1)^{f(1)}\begin{pmatrix}\frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}}\end{pmatrix}\right)\otimes\frac{1}{\sqrt{2}}(|0\rangle-|1\rangle)$$

$$|\psi_f\rangle = \frac{1}{\sqrt{2}}\left((-1)^{f(0)}\begin{pmatrix}\frac{1}{\sqrt{2}}\\ \frac{1}{\sqrt{2}}\end{pmatrix} + (-1)^{f(1)}\begin{pmatrix}\frac{1}{\sqrt{2}}\\ -\frac{1}{\sqrt{2}}\end{pmatrix}\right)\otimes\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

measure the 1st qubit of $|\psi_f\rangle$

if $f(0)=f(1)=0$
$f(0)=f(1)=1$ } constant functions — result $\pm\begin{pmatrix}1\\0\end{pmatrix} = \pm|0\rangle$

if $f(0)\neq f(1)=1$
$=0$ ~
$f(0)=1, f(1)=0$ } balanced functions — result $\pm\begin{pmatrix}0\\1\end{pmatrix} = \pm|1\rangle$

Unlike a classical computer,
only 1 evaluation done by
U_f to determine
whether f is a constant
function or a balanced function.