Great Ideas in Computing

University of Toronto CSC196
Fall 2021

Week 4: October 4-October 8 (2021)

Week 4 slides

Announcements:

- Assignment A1 is due this Friday, October 8 at 8AM on Markus. Please let me know if there is an issue downloading your submission. It should be a pdf document. There is a 5% penalty for each 24 hours the assignment is late with a maximum of 96 hours to submit a late assignment.
- :Next Wednesday, October 13, our second guest presenter Professor Nathan Wiebe, will discuss quantum computing. This will be an in-person class and I expect everyone to attend in person (unless there is a special reason why you cannot be there in which case you must tell me).

The agenda for this week

- I will finish up the discussion of the Dictionary data type and various data structures that can be used to implement a dictionary.
- The Church-Turing thesis (hypothesis) and a precise definition for the meaning of a **computable** function.

Dictionaries: A quick review and some additional data structures

The basic operations in a dictionary are the following:

- *Search*: Look up if someone is in the organization and if so retrieve the information for this person.
- *Update content*: Change the information regarding an item
- *Insert*: Add a new person to the organization
- *Delete*: Remove a person from the organization.

Sets of objects with these operations are referred to as a *Dictionary* data type. It is a static dictionary if we only want to look up and possibly modify records and a dynamic dictionary if we also want to add and delete. We can use different *data structures* to implement such a data type.

There can be many more operations that we want to perform on collections of data. More generally how one maintains and operates on data is known as the subfield of data bases. Analyzing data and extracting new (often statistical) information from collections of data is now called *data science* or *data analytics*. More ambitious learning of new information from data can be called *machine learning*.

Dictionaries lead to interesting concepts and ideas

- Many ways to implement a dictionary. What is important to note is that there are almost always TRADEOFFS in whatever we do in computing (and in life). How do you compare alternatives when there are multiple criteria for any given choice?. When can we say that choice 1 is better than choice 2 according to the given criteria?

Dictionaries lead to interesting concepts and ideas

- Many ways to implement a dictionary. What is important to note is that there are almost always TRADEOFFS in whatever we do in computing (and in life). How do you compare alternatives when there are multiple criteria for any given choice?. When can we say that choice 1 is better than choice 2 according to the given criteria?
- Here are some well known ways (called *data structures*) to implement a dictionary.
  1. An unordered list in an array
  2. An ordered list in an array
  3. A linked list
  4. A (balanced) search tree.
  5. A hash table.
- We will briefly talk about each of these possibilities. I do not want to get into details. Instead I just want to give a very high level idea of these different ways to implement a dynamic dictionary mentioning some tradeoffs and introducing some related concepts.

Brief discussion on these different methods

Let n be the current number of items in dictionary.

Each item has a unique name or *identifier*.

After I describe each method (on the white board), lets discuss some pros and cons of each method.

Some pros and cons of an unordered list in array for a dictionary

Relatively easy to add or delete an item (assuming we don't exceed the size of the array)

Some pros and cons of an unordered list in array for a dictionary

Relatively easy to add or delete an item (assuming we don't exceed the size of the array)

Requires an "average" of $n/2$ comparisons to find a current item and n comparisons to determine if the requested item is not in the current array. This is a hint of an important issue: namely, what does *average* mean?

Some pros and cons of an unordered list in array for a dictionary

Relatively easy to add or delete an item (assuming we don't exceed the size of the array)

Requires an "average" of $n/2$ comparisons to find a current item and n comparisons to determine if the requested item is not in the current array. This is a hint of an important issue: namely, what does *average* mean?

We usually have to indicate the size of the array in advance and would then have to allocate a new array if the number of entries exceeds the array size.

We need some memory management system for dynamic dictionaries. While this is true for any data structure, the onus is more on the algorithm designer for arrays..

Some pros and cons of an ordered list in an array

Note: This is only applicable if the items or the identifiers can be ordered which is usually the case.

Can search for an item in at most $\approx \log_2 n$ comparisons. Doing an asymptotic analysis of the time (and memory) for an algorithm is one of the main aspects in the analysis of an algorithm. Of course, correctness of the algorithm is paramount.

Some pros and cons of an ordered list in an array

Note: This is only applicable if the items or the identifiers can be ordered which is usually the case.

Can search for an item in at most $\approx \log_2 n$ comparisons. Doing an asymptotic analysis of the time (and memory) for an algorithm is one of the main aspects in the analysis of an algorithm. Of course, correctness of the algorithm is paramount.

$\log_2 n = x : 2^x = n$. Note that x will not be an integer unless $n = 2^k$ for some k.

To be precise the worst case number of comparisons is $\lfloor \log_2 n \rfloor + 1$ where the floor function is defined as $\lfloor x \rfloor =$ the largest integer $k \leq x$. You can verify that for $n = 2^k - 1$, the worst case number of comparison is k.

Ordered lists in an array continued

The differences between $\log n$ and n, can be dramatic (say if a search is within a *loop* of instructions). Even more dramatic is the difference between n and 2^n. We will be discussing further the importance of complexity issues.
It is more difficult to insert and delete records or modify the identifier of a record even for a fixed size array although updating the content of a record is easy once the item is accessed.

Can easily identify the i^{th} largest or smallest element.

And we usually have to specify the size of the array in advance.

Tables of some complexity bounding functions

Table 2

Polynomial-Time Algorithms Take Better Advantage of Computation Time

Time Complexity	n = 10	n = 20	n = 30	n = 40	n = 50	n = 60
n	0.00001 second	0.00002 second	0.00003 second	0.0000 second	0.00005 second	0.00006 second
n^2	0.0001 second	0.0004 second	0.0009 second	0.0016 second	0.0025 second	0.0036 second
n^3	0.001 second	0.008 second	0.027 second	0.064 second	0.125 second	0.216 second
n^5	0.1 second	3.2 seconds	24.3 seconds	1.7 minutes	5.2 minutes	13.0 minutes
2^n	0.001 second	1.0 second	17.9 minutes	12.7 days	35.7 years	366 centuries
3^n	0.059 second	58 minutes	6.5 years	3855 centuries	2×10^8 centuries	1.3×10^{13} centuries

Figure: Figure taken from Garey and Johnson "Computers and intractability : a guide to the theory of NP-completeness". Time in seconds based on an estimate of computers in the late 1970s. What if today computers are 100 times faster. Does this change the "message" in this figure.

A linked list

I may want to jump ahead to hash tables to motivate the exercises on Assignment A1.

Introduces the idea of a pointer

A linked list

I may want to jump ahead to hash tables to motivate the exercises on Assignment A1.

Introduces the idea of a pointer

I have shown a singly linked list. Can have a doubly linked list.

A linked list

I may want to jump ahead to hash tables to motivate the exercises on Assignment A1.

Introduces the idea of a pointer

I have shown a singly linked list. Can have a doubly linked list.

Easy to add items if the list is unordered. If list is ordered then have to follow pointers to see where to insert a new item.

A linked list

I may want to jump ahead to hash tables to motivate the exercises on Assignment A1.

Introduces the idea of a pointer

I have shown a singly linked list. Can have a doubly linked list.

Easy to add items if the list is unordered. If list is ordered then have to follow pointers to see where to insert a new item.

May have to traverse the entire list to find an item or determine it is not there.

A balanced binary search tree

A balanced binary tree with n "nodes" will have depth $\log_2 n$ and hence can search a balanced binary search tree in at most $\log_2 n$ "edge" traversals and comparisons.

I use the terminology of nodes and edges as a *tree* (in the sense of a search tree) is a special case of a *graph*. Graphs are also referred to as *networks* in many contexts (i.e. a social network, a transportation network, etc.).

The nodes (also called vertices) and edges (also called arcs in some applications) can be undirected or directed. In the latter case, we call a graph with directed edges a *directed graph* and usually mean an undirected graph if we just say graph.

We will be discussing further some graph concepts as the term progresses.

A hash table

We have a hash function $h : I \rightarrow M$ where $I = \{ID_1, \ldots ID_N\}$ is the set of all possible integer identifiers and $M = \{A[0], \ldots, A[m-1]\}$ is a small set of memory locations

That is, we are going to hash each of the $N = |I|$ possible items to a small set of $m = |M|$ memory locations.

Here we can have $N >> n$ where n is the actual number of items we are storing.
What is a suitable hash function h?

A hash table

We have a hash function $h : I \rightarrow M$ where $I = \{ID_1, \ldots ID_N\}$ is the set of all possible integer identifiers and $M = \{A[0], \ldots, A[m-1]\}$ is a small set of memory locations

That is, we are going to hash each of the $N = |I|$ possible items to a small set of $m = |M|$ memory locations.

Here we can have $N >> n$ where n is the actual number of items we are storing.
What is a suitable hash function h?
One possibility is $h(ID) = (a \cdot ID + b)(mod\ p)(mod\ m)$ where p is a large prime.

A hash table

Ignoring conflicts in the hash table, can search in constant time for a particular item

A hash table

Ignoring conflicts in the hash table, can search in constant time for a particular item

Need to deal with conflicts; i,.e multiple items hashing to the same place in the hash table. One possibility is to use a pointer to a linked list containing the IDs matched to the same place in the hash table.

A hash table

Ignoring conflicts in the hash table, can search in constant time for a particular item

Need to deal with conflicts; i,.e multiple items hashing to the same place in the hash table. One possibility is to use a pointer to a linked list containing the IDs matched to the same place in the hash table.
Nash tables introduce the use of probability, pseudo random numbers and pseudo random functions.

Note: When we draw random numbers in the execution of an algorithm, we are not drawing truly random numbers. The generation of pseudo random numbers and pseudo random functions is an interesting and substantial topic, one related to complexity theory, one of our future topics.

The birthday paradox

The birthday paradox: In probability theory, the birthday problem or birthday paradox concerns the probability that, in a "small" set of n randomly chosen people, some pair of them will have the same birthday with high probability.

By the pigeonhole principle, the probability reaches 100% when the number of people reaches 366 (since there are only 365 possible birthdays, excluding February 29).

However, 99.9% probability is reached with just 70 people, and 50% probability with 23 people. These conclusions are based on the assumption that each day of the year (excluding February 29) is equally probable for a birthday.
Should we try to see if there are two people in our class with the same birthday?

Concluding remark

The choice of any particular data structure or algorithm will usually depend on the application. For example, in choosing a data structure what operations are being done more often than others is an essential consideration. For example, even in a dynamic data structure, what if we rarely have to add or delete records?

What can't a computer do?

When we see the rather spectacular ways in which computer algorithms can perform, it is natural to ask whether or not there is anything that eventually we cannot do by computers.

Watching this evolution of computation and communication over say the last 80 years (since the earliest general purpose computers) and, in particular some of the most recent applications of machine learning, one can be forgiven for perhaps believing that there are no ultimate limitations.

But if we are going to ask about the limitations of computation in a precise way, well then we will need some mathematical framework.

That leads us to the seminal 1930s work of Alan Turing and independently Alonzo Church.

Computer Science as a mathematical science

David Hilbert was one of the great mathematicians of the late 19th and early 20th centuries. He asked the following question in **1900** known as Hilbert's 10^{th} problem:

"Given a Diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: To devise a process according to which it can be determined in a finite number of operations whether the equation is solvable in rational integers"

Here is a more familiar way to ask this question:

Given a polynomial $P(x_1, \ldots, x_n)$ with integer coefficients in many variables, decide if P has an integer root. That is, do there exist integers $i_1, \ldots, i_n$ such that $P(i_1, \ldots, i_n) = 0$?

As an example, $P(x) = x - 2$ clearly has an integer root whereas $P(x) = x^2 - 2$ does not have an integer (or rational) root.

What is computable? What is decidable?

Hilbert's question was essentially to ask if there is an algorithm that could decide whether or not a given multivariate polynomial has an integer root. Hilbert didn't mention the words "algorithm" or "computer" but he did articulate the need to solve the problem in a finite number of "steps".

Hilbert believed there was such a decision procedure but did not formalize what it meant to say that a problem solution is *computable*.

Terminology: If the problem is a decision problem (i.e., where the solution is to output YES or NO) then we usually say decidable rather than computable.

Following a series of intermediate results over 21 years, in 1970 Matiyasevich gave the first proof that Hilbert's 10th problem was undecidable (in a precise sense we will next discuss).

Note: The problem is decidable for polynomials in one variable.

End of Monday, October 4 class Start of Wednesday class

Announcements:

- Although I plan to do all office hours on zoom, if someone really wants to meet in person we can arrange that.
- Next Monday, no class
- Next Wednesday, Nathan Wiebe will discuss quantum computing.
- The facebook whistleblower was interviewed on the CBS show "60 Minutes". Here is a link to the interview. We will come back to the issue of the spread of information and mis-information later in the term.
  https://www.cbsnews.com/news/facebook-whistleblower-frances-haugen-misinformation-public-60-minutes-2021-10-03/

Todays agenda

- I want to clarify comments I made about using two hash functions. We can apply a second hash function to hopefully resolve conflicts from the first hash function. When throwing n balls at n bins, the expected size of a bin is 1, but the expected size of the maximum loaded bin is roughly $\frac{\log n}{\log \log n}$. If each person (in sequence) throws two balls at the bins and places a ball in the currently least loaded bin then the expected size of the mazimum bin will be roughly $\log \log n$. This would not help us say in the dictionary application. Why?
- We ended Mondays class at slide 18 having just introduced Hilbert's 10th problem and stated the result (i.e., mathematical theorem) that the problem was undecidable.
- But, of course, we are jumping ahead (on purpose for motivation) as we do not yet have a precise definition for "computable". And this is where we begin today.

A precise definition for the meaning of "decidable"

We have studied the von Neumann model as a model of computation but we never gave a precise definition but more or less relied on our prior knowledge of how we think computers work. And we didn't give a definition for what is an *algorithm*.

Computers are continually getting faster and have larger memories so must our concept of what is computable also be constantly changing? Could Hilbert's problem become decidable tomorrow?

We also briefly touched upon the complexity of operations with respect to the data structures for the dictionary data type. Must the complexity of operations and the complexity of algorithms also change constantly?

This raises a fundamental question: Is there an ultimate precise model of computation with respect to which we would then have a precise meaning of a computable function? Or must we continually be changing our understanding of what is and what is not computable?

A precise definition for the meaning of computable (decidable) continued

High level models such as the von Neumann model provide a good intuition for what we have in mind when we say a function f is decidable. **But we really need a precise mathematical model if we want to prove mathematical results.**

Independently in 1936, Alonzo Church and Alan Turing published formal definitions for what it meant to be computable. These papers were very influential for the von Neumann model which comes about 10 years later.

Church's definition was based on a formalism in logic called the lambda calculus where one starts with some basic functions and then specifies ways to compose new functions from existing functions.

Alan Turing proposed a precise model of computation which we will only briefly describe. Turing also went on to show that these two very different models are provably equivalent in the sense that they result in the same set of computable functions.

But are there other models?

For a number of years other models were considered and all turn out to be equivalent (and sometimes weaker) than the Church-Turing models.

This led to the following Church-Turing hypothesis. Every plausible model of computation is equivalent to (or weaker than) Turing's very basic computational model. This is not to say that Turing machines are as easy to program or will lead to the same complexity analysis. **But the meaning of computable does not change.**

In particular, what about quantum computing?

But are there other models?

For a number of years other models were considered and all turn out to be equivalent (and sometimes weaker) than the Church-Turing models.

This led to the following Church-Turing hypothesis. Every plausible model of computation is equivalent to (or weaker than) Turing's very basic computational model. This is not to say that Turing machines are as easy to program or will lead to the same complexity analysis. **But the meaning of computable does not change.**

In particular, what about quantum computing? It could very well be that quantum computing will substantially change our sense of what is "efficiently computable" but it does not enlarge the meaning of "computable".

Note: The Church Turing hypothesis in NOT a theorem. It is an almost universally believed statement about the nature of digital computing. Could someday we come to believe that there are more inclusive models?

But are there other models?

For a number of years other models were considered and all turn out to be equivalent (and sometimes weaker) than the Church-Turing models.

This led to the following Church-Turing hypothesis. Every plausible model of computation is equivalent to (or weaker than) Turing's very basic computational model. This is not to say that Turing machines are as easy to program or will lead to the same complexity analysis. **But the meaning of computable does not change.**

In particular, what about quantum computing? It could very well be that quantum computing will substantially change our sense of what is "efficiently computable" but it does not enlarge the meaning of "computable".

Note: The Church Turing hypothesis in NOT a theorem. It is an almost universally believed statement about the nature of digital computing. Could someday we come to believe that there are more inclusive models? Yes but so far our experience leads us to believe that the hypotheis will continue to be (almost) universally accepted.

Comments on Turing's model

- We can assume there is a halting state q_{halt} such that the machine halts if it enters state q_{halt}. There is also an initial state q_0.
- We view a Turing machine P as computing a function $f_P : \Sigma^* \to \Sigma^*$ where $\Sigma \subseteq \Gamma$ where $y = f(x)$ is the string that remains if (and when) the machine halts. There can be other conventions as to interpreting the resulting output y.
- Note that the model is precisely defined and the definition of a computation step is also precise. (See slides 26 and 27.)
- For decision problems, we can have two halting states, a YES and NO state.

More about Turing's seminal results

- Turing showed that there is a Universal Turing machine (UTM) call it U. That is, given an input $p\#x$ the machine interprets the string p as a Turing machine description (i.e. as a state transition function δ) and $x \in \Sigma^*$ is interpreted as the input to the machine P described by p and $f_U(p\#x) = f_P(x)$.
- In modern terms, a UTM is an *interpreter*.
- Turing showed that the **halting problem** is undecidable. That is, there does not exist a fixed TM F such that F when executed on an input string $(p\#x)$, where p encodes a TM P, whether or not P will halt and correctly decide if P halts on the input string x. It is also undecidable if a TM P will halt on all inputs.
- As a consequence, this means that you cannot have a compiler which will check for the algorithm $\mathcal{A}$ you have written whether or not $\mathcal{A}$ will halt on every input.

End of Wednesday, October 6 class

We ended just having stated the fact that (given the Church-Turing hypothesis), it is undecidable to determine if a computer program (given as an input) will halt on a specific input. It is also undecidable if a program will halt on all inputs. Of course, for a specific program we may be able to reason that it will halt on a given input or on all inputs.

From these basic undecidablitliy results follow many other problems that are undecidable. In the remaining slides for this week I am mentioning one classic undecidability result answering another question by Hilbert.

On Monday, October 18, I will complete our introduction to computability and then mention briefly a formalization of "efficiently computable". Then (depending on the time) move on to a new more familiar topic, namely search engines.

The Entscheidungsproblem

In his seminal paper "On Computable Numbers With an Application to the Entscheidungsproblem (i.e. decision problem), Turing uses his model and the undecidability of the halting problem, to prove the undecidabiliy of the "Entscheidungsproblem" posed by Hilbert in 1928. (Church provided an independent proof within his formalism.)

Sometimes this is informally stated as "can mathematics be decided ?"

The Entscheidungsproblem question refers to the decidability of predicate logic which Church and Turing independently resolved in 1936-1937. It would take a little while to formally define this "Entscheidungsproblem" but here is an example of the kind of question that one wants to answer:
Given a formula such as $\forall x \exists y : y < x$
can we determine if such a formula is always true no matter what what ordered domain x, y and $<$ refer to?
For example, $x < y$ and $y < z$ implies $x \neq z$ is always true.
But $x < y$ implies $\exists z : x < z < y$ is not true of all ordered domains (e.g., consider the integers)

A pictorial representation of a Turing machine

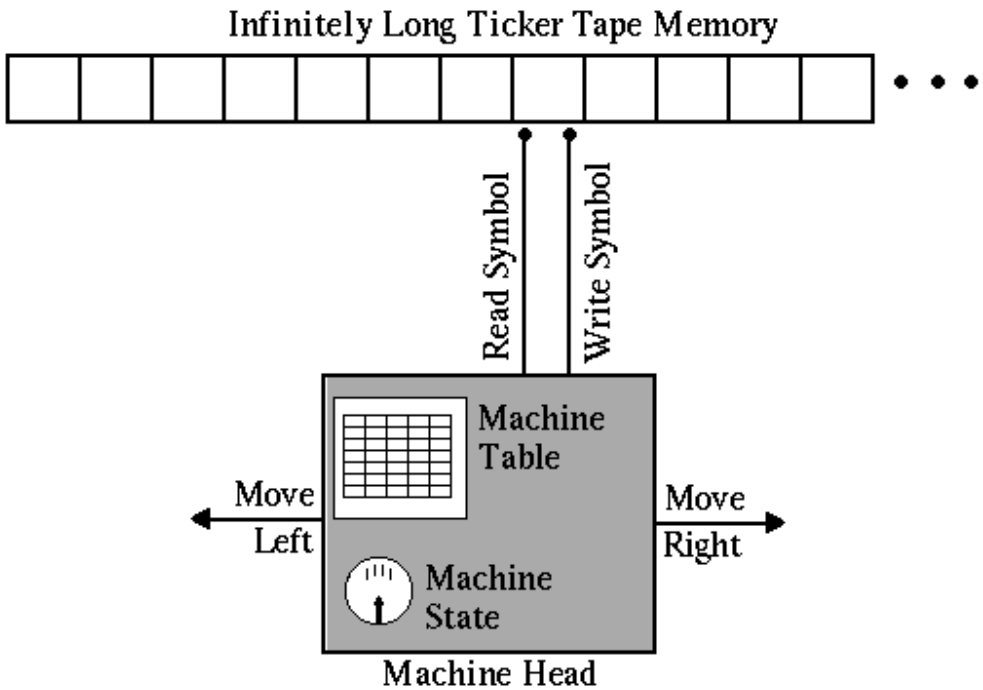

Figure: Figure taken from Michael Dawson "Understanding Cognitive Science"

Comments on Turing's model

- Formally, a Turing machine algorithm is described by the following function $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$
  Q is a *finite* set of *states*. Γ is a finite set of symbols
  (e.g., $\Gamma = \{\#, 0, 1, a, b, \ldots\}$ and perhaps $\Sigma = \{0, 1\}$)
- Note: Each δ function is the definition of a single Turing machine; that is, each δ function is the statement of an algorithm.
- We can assume there is a halting state q_{halt} such that the machine halts if it enters state q_{halt}. There is also an initial state q_0.
- We view a Turing machine P as computing a function $f_P : \Sigma^* \rightarrow \Sigma^*$ where $\Sigma \subseteq \Gamma$ where $y = f(x)$ is the string that remains if (and when) the machine halts. There can be other conventions as to interpreting the resulting output y.
- Note that the model is precisely defined as is the concept of a computation step. A *configuration* of a TM is specified by the contents of the tape, the state, and the position of the tape head. A computation of a TM is a sequence of configurations, starting with an initial configuration.
- For decision problems, we can have YES and NO halting states.

A more general Turing machine model

The Turing machine model has been extended to allow separate read (for the input) and write (for the output when computing a function) tapes and any finite number of work tapes. Here is a figure of a multi-tape TM (but without separate input and output tapes).

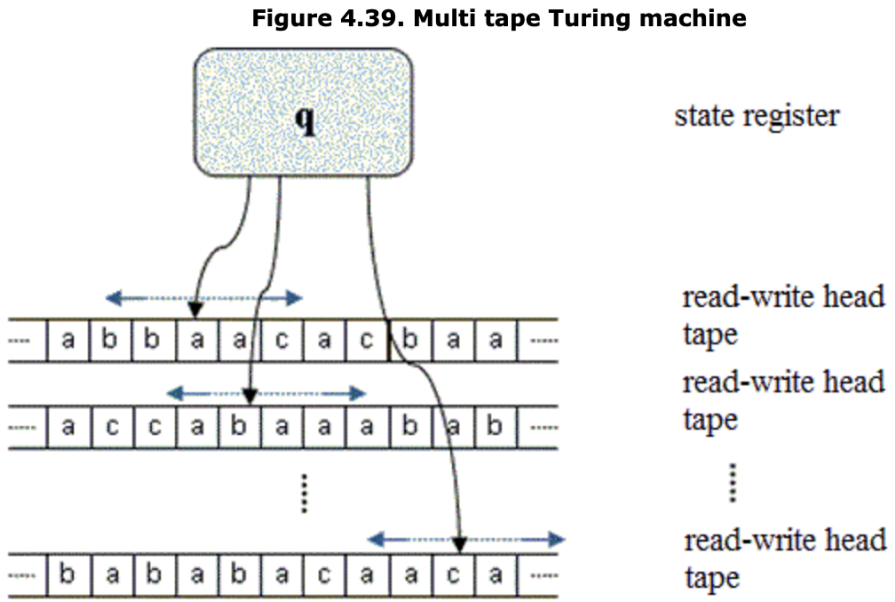

Figure: Figure from the Bela Gyires Informatics Curriculum Repository

The extended Church-Turing Thesis

Church-Turing computability provides a formal definition of *computable* (in principle). But it allows for computable functions of arbitrary complexity.

Indeed it is not difficult to show (again using diagonalization) that for any time bound $T(n)$ there are computable decision problems that require more time (or more memory) than $T(n)$ for sufficiently large values of n where n is the length of the input and output strings. Why do we usually say that operations in the von Neumann model take unit time?.

The extended Church-Turing Thesis

Church-Turing computability provides a formal definition of *computable* (in principle). But it allows for computable functions of arbitrary complexity.

Indeed it is not difficult to show (again using diagonalization) that for any time bound $T(n)$ there are computable decision problems that require more time (or more memory) than $T(n)$ for sufficiently large values of n where n is the length of the input and output strings. Why do we usually say that operations in the von Neumann model take unit time?.

The extended Church-Turing thesis states that any function that is "**feasibly computable**" must be computable by a Turing machine within time $p(n)$ where $p()$ is a polynomial. This extended thesis is stated with regard to classical computers and not necessarily quantum computers.

Note: Currently we (in theoretical CS) believe that there are functions (e.g., factoring large integers) "efficiently computable" by quantum computers but not efficiently computable by classical computers. . But can quantum computers of sufficient size be built?

Extended Church-Turing thesis continued

Informal theorem: Any "**reasonable"** classical computation model $\mathcal{M}$ can be simulated by a one tape or multi-tape Turing machine so that if say f is computable in time $T(n)$ on $\mathcal{M}$ then f is computable on a Turing machine in time $p_{\mathcal{M}}(T(n))$ for some fixed polynomial $p_{\mathcal{M}}$.

Using a mutli-tape TM, for most models, $p_{\mathcal{M}}(m)$ is $O(m^2)$ or $O(m^3)$. For example, if $T(n)$ is n^2 and $p_{\mathcal{M}}(m)$ is m^3 then $p_{\mathcal{M}}(T(n)$ is n^6.

What is **"not reasonable"**?

Extended Church-Turing thesis continued

Informal theorem: Any "**reasonable**" classical computation model $\mathcal{M}$ can be simulated by a one tape or multi-tape Turing machine so that if say f is computable in time $T(n)$ on $\mathcal{M}$ then f is computable on a Turing machine in time $p_{\mathcal{M}}(T(n))$ for some fixed polynomial $p_{\mathcal{M}}$.

Using a mutli-tape TM, for most models, $p_{\mathcal{M}}(m)$ is $O(m^2)$ or $O(m^3)$. For example, if $T(n)$ is n^2 and $p_{\mathcal{M}}(m)$ is m^3 then $p_{\mathcal{M}}(T(n)$ is n^6.

What is **"not reasonable"**?

Consider having unit time operations $+, -, *, \div$ where $\div$ means integer division; that is, dividing a by b results in the remainder. Now consider repeated squaring of $2, 2^2 = 4, 4^2 = 16, \ldots, 2^{(2^n)}$. That is, in n multiplications, we can construct an integer whose binary representation has 2^n bits.

It turns out that with such a model we can factor integers in polynomial time. **BUT** this is not reasonable as we are doing classical operations on exponentially long strings in unit time which is not reasonable.