AAiT
Addis Ababa Institute of Technology
አዲስ አበባ ቴክኖሎጂ ኢንስቲትዩት
Addis Ababa University
አዲስ አበባ ዩኒቨርሲቲ

Chapter 1

Introduction to Algorithms

Yonas Y.
AAiT/SECE

1. The Role of Algorithms in Computing
   - Algorithms
   - Algorithms as a technology
2. The Sorting Problem
   - Insertion Sort
   - Analyzing algorithms
   - Designing algorithms
3. Growth of Functions
   - Asymptotic notation
4. Divide-and-Conquer
   - The maximum-subarray problem
   - Strassen's algorithm for matrix multiplication
   - The substitution method for solving recurrences
   - The recursion-tree method for solving recurrences
   - The master method for solving recurrences
5. Probabilistic Analysis and Randomized Algorithms
   - The hiring problem
   - Indicator random variables
   - Randomized algorithms

Algorithms

- Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.

- An algorithm is thus a sequence of computational steps that transform the input into the output.

- We can also view an algorithm as a tool for solving a well-specified computational problem.

Algorithms

The algorithm describes a specific computational procedure for achieving that input/output relationship.

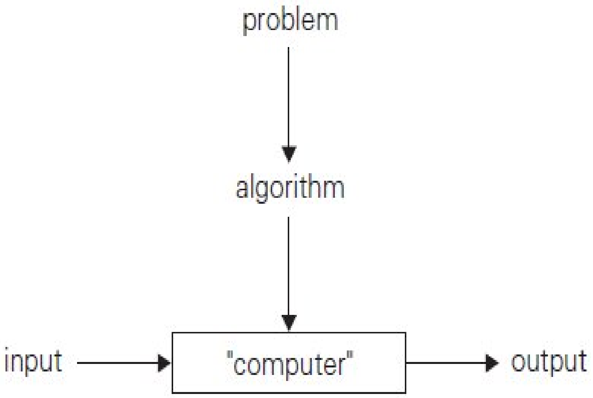


Figure: 1 The notion of the algorithm.

For example, we might need to sort a sequence of numbers into non-decreasing order (sorting problem).

- **Input**: A sequence of n numbers $[a_1, a_2, ..., a_n]$.
- **Output**: A permutation (reordering) $[a_1^{'}, a_2^{'}, ..., a_n^{'}]$ of the input sequence such that $[a_1^{'} \leq a_2^{'}, \leq, ..., \leq, a_n^{'}]$.

For a given input sequence of [31, 41, 59, 26, 41, 58], a sorting algorithm returns as output the sequence [26, 31, 41, 41, 58, 59].

- Such an input sequence is called an instance of the sorting problem.
- An algorithm is said to be correct if, for every input instance, it halts with the correct output.

What kinds of practical problems are solved by algorithms?

Figure: 2 The Internet

Figure: 3 E-Commerce

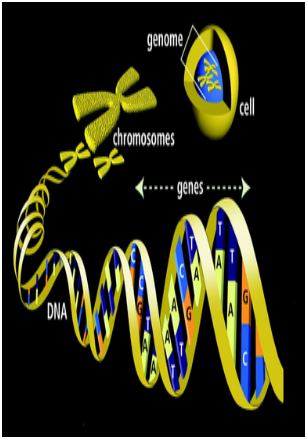


Figure: 4 The Human Genome Project

Algorithms

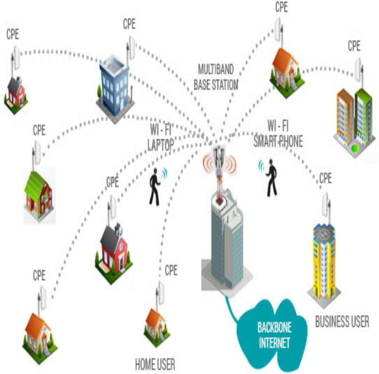


Figure: 5 Internet Service Providers

Figure: 6 Industries

Algorithms as a technology

Algorithms depend on

- Computational capacity of computers
- Memory

So these resources should be used wisely.

Algorithms efficiency = Computational time + Space + . . .

Efficiency

- Different algorithms devised to solve the same problem often differ dramatically in their efficiency.
- These differences can be much more significant than differences due to hardware and software.
- Therefore, total system performance depends on choosing efficient algorithms as much as on choosing fast hardware.

Example

Insertion Sort $\Longrightarrow$ $c_1 \cdot n^2$

Merge Sort $\Longrightarrow$ $c_2 \cdot n\,lg\,n$ where $lg\,n = log_2\,n$

Let us pit a faster computer (computer A) running insertion sort against a slower computer (computer B) running merge sort.

- Let each must sort an array of 10 million numbers.

Suppose:

- Computer A executes 10 billion instructions per second.
- Computer B executes only 10 million instructions per second.

To make the difference even more dramatic, let $c_1 = 2$ and $c_2 = 50$.

To sort the 10 million numbers:

Computer A $\Longrightarrow$ $\dfrac{2\cdot(10^7)^2 \ \textit{instructions}}{10^{10} \ \textit{instructions/second}} = 20{,}000$ seconds (more than 5.5 hours).

Computer B $\Longrightarrow$ $\dfrac{50\cdot 10^7 \lg 10^7 \ \textit{instructions}}{10^7 \ \textit{instructions/second}} \approx 1163$ seconds (less than 20 minutes).

Algorithms and other technologies

Change in computer technologies $\Longleftrightarrow$ rapid changes in algorithms.

Even an application that does not require algorithmic content at the application level relies heavily upon algorithms.

- The hardware design used algorithms.
- The design of any GUI relies on algorithms.
- Routing in networks relies heavily on algorithms.

Insertion Sort

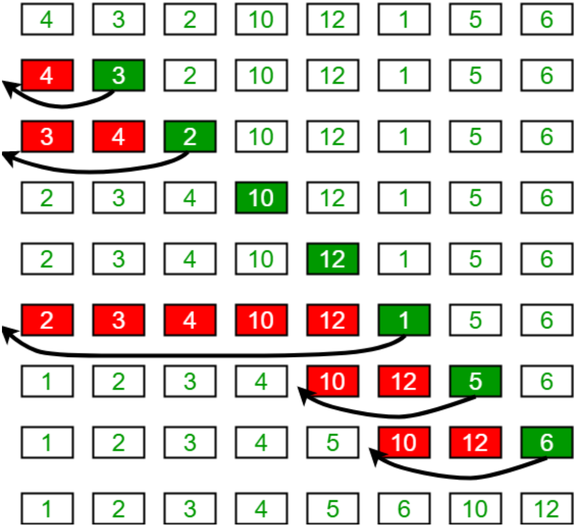
This algorithm solves the sorting problem introduced in the previous section.

- **Input**: A sequence of n numbers $[a_1, a_2, \ldots, a_n]$.
- **Output**: A permutation (reordering) $[a_1^{'}, a_2^{'}, \ldots, a_n^{'}]$ of the input sequence such that $[a_1^{'} \leq a_2^{'}, \leq, \ldots, \leq, a_n^{'}]$.

The numbers that we wish to sort are also known as the keys.

- Insertion sort works the same way many people sort a hand of *playing cards*.
- The algorithm sorts the input numbers in place: it rearranges the numbers within the original array.

Insertion Sort



4 3 2 10 12 1 5 6

4 3 2 10 12 1 5 6

3 4 2 10 12 1 5 6

2 3 4 10 12 1 5 6

2 3 4 10 12 1 5 6

2 3 4 10 12 1 5 6

1 2 3 4 10 12 5 6

1 2 3 4 5 10 12 6

1 2 3 4 5 6 10 12

Figure: 7 The operation of INSERTION-SORT on the array A = [4, 3, 2, 10, 12, 1, 5, 6].

Algorithm 1 INSERTION-SORT(A)

```
1:  for $j = 2$ to $A.length$ do
2:      $key = A[j]$
3:      // Insert $A[j]$ into the sorted sequence $A[1, \ldots, j-1]$.
4:      $i = j - 1$
5:      while $i > 0$ and $A[i] > key$ do
6:          $A[i+1] = A[i]$
7:          $i = i - 1$
8:      end while
9:      $A[i+1] = key$
10: end for
```

- At the start of each iteration of the for loop of lines 1-9, the subarray $A[1, \ldots, j-1]$ consists of the elements originally in $A[1, \ldots, j-1]$, but in sorted order.

We state these properties of $A[1, \ldots, j-1]$ formally as a loop invariant.

Loop invariants help us to understand why an algorithm is correct.

- Initialization: It is true prior to the first iteration of the loop.
- Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.
- Termination: When the loop terminates, the invariant gives us a useful property that helps to show that the algorithm is correct.

Let us see how these properties hold for insertion sort.

- **Initialization**: Start by showing that the loop invariant holds before the first loop iteration, when $j = 2$.
    - The subarray $A[1, \ldots, j-1]$, therefore, consists of just the single element $A[1]$.

- **Maintenance**: Informally, the body of the **for** loop works by moving $A[j-1]$, $A[j-2]$, $A[j-3]$, and so on by one position to the right until it finds the proper position for $A[j]$.

- **Termination**: The condition causing the **for** loop to terminate is that $j > A.length = n$.

Analyzing algorithms

Analyzing an algorithm $\Longrightarrow$ predicting the resources that the algorithm requires.

- Memory, communication bandwidth, or computer hardware $\Longrightarrow$ **Computational time**.
- Generally, by analyzing several candidate algorithms for a problem, we can identify a most efficient one.
- We will use RAM model for analysis in this course.
  - generic one processor,
  - no concurrent operations,
  - algorithms will be implemented as computer programs.

Analysis of insertion sort

The time taken by the INSERTION-SORT procedure depends:

- Size of input,
- How nearly sorted they already are.

In general, the time taken by an algorithm grows with the size of the input.

- running time of a program $\Longleftrightarrow$ f (size of its input).

Input size depends on the problem being studied and can take the following forms:

- number of items in the input
- total number of bits needed to represent the input
- numbers of vertices and edges in graphs

The **running time** of an algorithm on a particular input is the number of primitive operations or "steps" executed.

- Assume that each execution of the i^{th} line takes time c_i, where c_i is a constant.

Analyzing the running time of INSERTION-SORT.

```
Algorithm 1 INSERTION-SORT(A)                      Cost       times
for $j = 2$ to $A.length$ do                       ▷ $c_1$    $n$
    $key = A[j]$                                   ▷ $c_2$    $n-1$
    // Insert $A[j]$ into the sorted ...           ▷ 0        $n-1$
    $i = j - 1$                                    ▷ $c_4$    $n-1$
    while $i > 0$ and $A[i] > key$ do              ▷ $c_5$    $\sum_{j=2}^{n} t_j$
        $A[i+1] = A[i]$                            ▷ $c_6$    $\sum_{j=2}^{n}(t_j - 1)$
        $i = i - 1$                                ▷ $c_7$    $\sum_{j=2}^{n}(t_j - 1)$
    end while
    $A[i+1] = key$                                 ▷ $c_8$    $n-1$
end for
```

To compute $T(n)$, the running time of INSERTION-SORT on an input of n values:

- Sum the products of the cost and times columns, obtaining

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1)$$

Even for inputs of a given size, an algorithm's running time may depend on which input of that size is given.

The Best-case running time

- For each $j = 2, 3, \ldots, n$, we then find that $A[i] \leq key$ in line 5 when i has its initial value of $j - 1$.
- Thus $t_j = 1$ for $j = 2, 3, \ldots, n$, and the best-case running time is

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \end{aligned}$$

We can express this running time as $an + b$ for constants a and b that depend on the statement costs c_i; thus a linear function of n.

The worst case happens, if

- The array is in reverse sorted order-that is, in decreasing order
- We must compare each element $A[j]$ with each element in the entire sorted subarray $A[1, ..., j-1]$, and so $t_j = j$ for $j = 2, 3, ..., n$.

Notice that

$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1,$ and

$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$

We find that in the worst case, the running time of INSERTION-SORT is

$$
\begin{aligned}
T(n) = {} & c_1 n + c_2(n-1) + c_4(n-1) + c_5(\frac{n(n+1)}{2} - 1) \\
& + c_6(\frac{n(n-1)}{2}) + c_7(\frac{n(n-1)}{2}) + c_8(n-1) \\
& = (\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2})n^2 + (c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8)n \\
& - (c_2 + c_4 + c_5 + c_8)
\end{aligned}
$$

We can express this worst-case running time as $an^2 + bn + c$ for constants a, b, and c that again depend on the statement costs c_i ; it is thus a quadratic function of n.

Worst-case and Average-case analysis

In analyzing algorithms we are often interested in the worst-case running time, that is, the longest running time for any input of size n.

Some of the reasons behind this are:

- The worst-case running time of an algorithm gives us an upper bound on the running time for any input.
- For some algorithms, the worst case occurs fairly often. Like, searches for absent information from a database.
- The ”average case” is often roughly as bad as the worst case.

Order of growth

For the worst-case running time of INSERTION-SORT $an^2 + bn + c$:

- The rate of growth, or order of growth mainly depends on the leading term of a formula (e.g., an^2).
- We also ignore the leading term's constant coefficient.

For insertion sort, computational efficiency depends on the factor of n^2 from the leading term.

- We write that insertion sort has a worst-case running time of $\Theta(n^2)$.

The divide-and-conquer approach

Many useful algorithms are recursive in structure.

- These algorithms typically follow a divide-and-conquer approach.

The divide-and-conquer paradigm involves three steps at each level of the recursion:

- Divide the problem into a number of subproblems that are smaller instances of the same problem.
- Conquer the subproblems by solving them recursively.
- Combine the solutions to the subproblems into the solution for the original problem.

The merge sort algorithm closely follows the divide-and-conquer paradigm.

- **Divide**: Divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each.
- **Conquer**: Sort the two subsequences recursively using merge sort.
- **Combine**: Merge the two sorted subsequences to produce the sorted answer.

The recursion "bottoms out" when the sequence to be sorted has a length of 1.

Designing algorithms

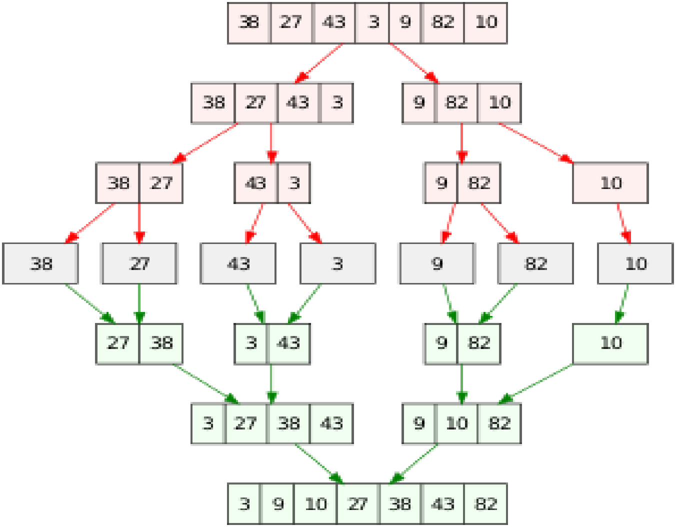


Figure: 8 The operation of merge sort on the array.

- The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step.
- We merge by calling an auxiliary procedure `MERGE(A, p, q, r)`, where A is an array and p, q, and r are indices into the array such that $p \leq q < r$.
- The procedure assumes that the subarrays $A[p..q]$ and $A[q+1..r]$ are in sorted order.
- It merges them to form a single sorted subarray that replaces the current subarray $A[p..r]$.
- The MERGE procedure takes time $\Theta(n)$, where $n = r - p + 1$ is the total number of elements being merged.

Algorithm 2 MERGE(A, p, q, r)

```
$n_1 = q - p + 1$
$n_2 = r - q$
// let $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$ be new arrays
for $i = 1$ to $n_1$ do
    $L[i] = A[p + i - 1]$
end for
for $j = 1$ to $n_2$ do
    $R[j] = A[q + j]$
end for
$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
$i = 1$
$j = 1$
for $k = p$ to $r$ do
    if $L[i] \leq R[j]$ then
        $A[k] = L[i]$
        $i = i + 1$
    else
        $A[k] = R[j]$
        $j = j + 1$
    end if
end for
```

The above pseudocode perform the $r - p + 1$ basic steps by maintaining the following loop invariant:

- At the start of each iteration of the **for** loop of lines 14-22, the subarray $A[p..k-1]$ contains the $k - p$ smallest elements of $L[1..n_1+1]$ and $R[1..n_2+1]$, in sorted order.
- Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.

The procedure `MERGE-SORT(A,p,r)` sorts the elements in the subarray $A[p..r]$.

1. If $p \geq r$, the subarray has at most one element and is therefore already sorted.
2. Otherwise, the divide step simply computes an index q that partitions $A[p..r]$ into two subarrays:

Algorithm 2 MERGE-SORT(A, p, r)

```
if $p < r$ then
    $q = \lfloor (p + r)/2 \rfloor$
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q+1, r)
    MERGE(A, p, q, r)
end if
```

Analyzing divide-and-conquer algorithms

When an algorithm contains a recursive call to itself,

- Describe its running time by a recurrence equation or recurrence.
- If the problem size is small enough, say $n \leq c$ for some constant c, the straightforward solution takes constant time, which we write as $\Theta(1)$.
- If the problem yields a subproblems, each of which is 1/b the size of the original, it takes time aT(n/b) to solve **a** of them.

If

- We take $D(n)$ time to divide the problem into subproblems and
- $C(n)$ time to combine the solutions to the subproblems into the solution to the original problem

The recurrence will be

$$T(n) = \begin{cases} \Theta(1), & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n), & \text{otherwise} \end{cases}$$

Analysis of merge sort

When we have $n > 1$ elements, we break down the running time as follows.

- Divide: The divide step just computes the middle of the subarray $\implies D(n) = \Theta(1)$.
- Conquer: We recursively solve two subproblems, each of size $n/2 \implies 2T(n/2)$.
- Combine: $C(n) = \Theta(n)$.

Therefore, the total running time of the MERGE-SORT algorithm is:

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1 \\ 2T(n/2) + \Theta(n), & \text{if } n > 1 \end{cases}$$

Asymptotic notation

Discusses how the running time of an algorithm increases as the size of the input increases without bound.

- The notation domains are the set of natural numbers $\mathbb{N} = \{0, 1, 2, ...\}$,
- Such notations are convenient for describing the worst-case running-time function $T(n)$.
- For *insertion sort*:- worst case running time is $an^2 + bn + c \implies \Theta(n^2)$.

Θ-notation

For a given function $g(n)$, we denote by $\Theta(g(n))$ - notation the *set of functions*:

$$\Theta(g(n)) = \{f(n): \texttt{ there exist positive constants } c_1, c_2\texttt{, and } n_0 \texttt{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \texttt{ for all } n \geq n_0\}\texttt{.}$$

- A function $f(n)$ belongs to the set $\Theta(g(n))$ if there exist positive constants c_1 and c_2 such that it can be "sandwiched" between $c_1 g(n)$ and $c_2 g(n)$, for sufficiently large n.

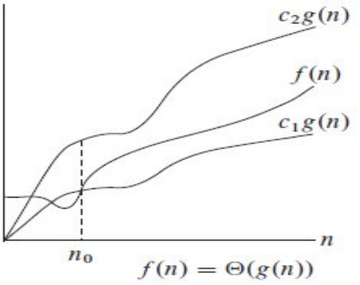


Figure: 9 Θ-notation bounds a function to within constant factors.

- We say that $g(n)$ is an asymptotically tight bound for $f(n)$.
- $f(n) \in \Theta(g(n))$ should be asymptotically nonnegative.

Example: Show that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$?

Using the formal asymptotic definition,

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2, \qquad \text{for all } n \geq n_0.$$

Dividing by n^2 yields

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

- The right-hand inequality hold for any value $n \geq 1$ by choosing any constant $c_2 \geq 1/2$.
- Likewise, we can make the left-hand inequality hold for any value of $n \geq 7$ by choosing any constant $c_1 \leq 1/14$.

Thus, by choosing $c_1 = 1/14$, $c_2 = 1/2$, and $n_0 = 7$, we can verify that $\frac{1}{2}n^2 - 3n = \Theta(n^2)$.

O-notation

When we have only an asymptotic upper bound, we use O-notation.

- For a given function $g(n)$, we denote by $O(g(n))$-notation the *set of functions*:

$$O(g(n)) = \{f(n) : \texttt{ there exist positive constants } c \texttt{ and } n_0 \texttt{ such that } 0 \leq f(n) \leq cg(n) \texttt{ for all } n \geq n_0\}.$$

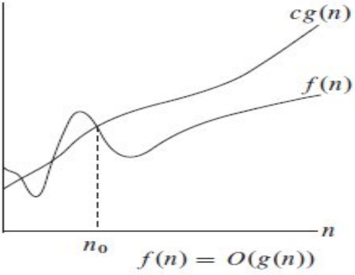


Figure: 10 O-notation gives an upper bound for a function to within a constant factor.

- Note that $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$, since Θ-notation is a stronger notion than O-notation.
- Since O-notation describes an upper bound, when we use it to bound the worst-case running time of an algorithm.

Ω-notation

The Ω-notation provides an asymptotic lower bound.

- For a given function $g(n)$, we denote by $\Omega(g(n))$-notation the *set of functions*:

$$\Omega(g(n)) = \{f(n) : \texttt{ there exist positive constants } c \texttt{ and } n_0 \texttt{ such that } 0 \leq cg(n) \leq f(n) \texttt{ for all } n \geq n_0\}.$$

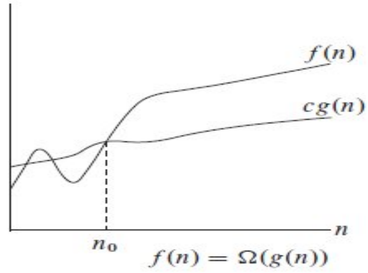


Figure: 11 Ω-notation gives a lower bound for a function to within a constant factor.

Theorem 1.1:

For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Divide-and-Conquer

- In divide-and-conquer, apply three steps at each level of the recursion $\Longrightarrow$ [`Divide`, `Conquer` `and` `Combine`].
- When the subproblems are large enough to solve recursively, we call that the recursive case.
- Once the subproblems become small enough that we no longer recurse, we say that the recursion "bottoms out" and that we have gotten down to the base case.

Recurrences

- A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.

For example:

A recursive algorithm might divide subproblems into unequal sizes, such as a 2/3-to-1/3 split.

- If the divide and combine steps take linear time, such an algorithm would give rise to the recurrence

$$T(n) = T(2n/3) + T(n/3) + \Theta(n).$$

Three methods are discussed for solving recurrences - that is, for obtaining asymptotic "Θ" or "O" bounds on the solution:

- In the substitution method, we guess a bound and then use mathematical induction to prove our guess correct.
- The recursion-tree method converts the recurrence into a tree whose nodes represent the costs incurred at various levels of the recursion.
- The master method provides bounds for recurrences of the form

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

  where $a \geq 1$, $b > 1$, and $f(n)$ is a given function.

Occasionally, we shall see recurrences that are not equalities but rather inequalities,

- such as $T(n) \leq 2T(n/2) + \Theta(n)$ (will be represented by $O - notation$)
- $T(n) \geq 2T(n/2) + \Theta(n)$ (will be represented by $\Omega - notation$).

When we state and solve recurrences, we often omit floors, ceilings, and boundary conditions.

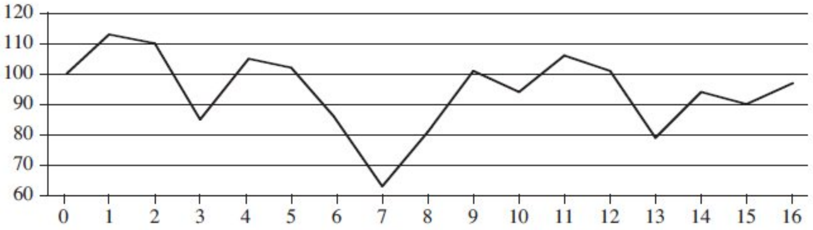
- For any real number x, we denote the greatest integer less than or equal to x by $\lfloor x \rfloor$ (read "the floor of x") and the least integer greater than or equal to x by $\lceil x \rceil$ (read "the ceiling of x").
- Omit statements of the boundary conditions of recurrences and assume that $T(n)$ is constant for small n.

The maximum-subarray problem

Suppose that you been offered the opportunity to invest in **x software and app. development company**.

- You are allowed to buy one unit of stock only one time and then sell it at a later date,
- Buying and selling is performed after the close of trading for the day.
- To compensate for this restriction, you are allowed to learn what the price of the stock will be in the future.
- Your goal is to maximize your profit.

The maximum-subarray problem

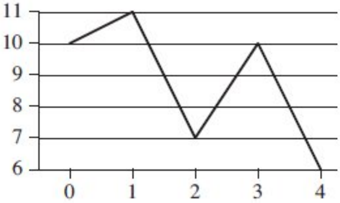


Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

Figure: 12 Information about the price of stock in the x software and app. development company after the close of trading over a period of 17 days. The horizontal axis of the chart indicates the day, and the vertical axis shows the price. The bottom row of the table gives the change in price from the previous day.

The straightforward solution to maximize profit is buying when the stock price is low and selling when its high.

This might not always be the feasible solution as shown in figure 13.



Day	0	1	2	3	4
Price	10	11	7	10	6
Change		1	−4	3	−4

Figure: 13 An example showing that the maximum profit does not always start at the lowest price or end at the highest price.

A brute-force solution

We can easily devise a brute-force[1] solution to this problem:

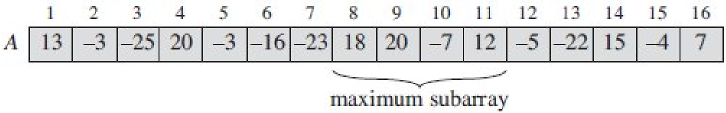
- Try every possible pair of buy and sell dates in which the buy date precedes the sell date.
- A period of n days has $\binom{n}{2}$ such pairs of dates.
- Since $\binom{n}{2}$ is $\Theta(n^2)$, and the best we can hope for is to evaluate each pair of dates in constant time, this approach would take $\Omega(n^2)$ time.

[1] Brute force may refer to any of several problem-solving methods involving the evaluation of multiple (or every) possible answer(s) for fitness.

A transformation

Instead of looking at the daily prices,

- let us instead consider the daily change in price, where the change on day i is the difference between the prices after day $i - 1$ and after day i.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	–3	–25	20	–3	–16	–23	18	20	–7	12	–5	–22	15	–4	7

maximum subarray

Figure: 14 The change in stock prices as a maximum-subarray problem. Here, the subarray A[8,. . . , 11], with sum 43, has the greatest sum of any contiguous subarray of array A.

A solution using divide-and-conquer

Suppose we want to find a maximum subarray of the subarray `A[low,...,high]`.

- Divide-and-conquer suggests that we divide the subarray into two subarrays of as equal size as possible.
- Find the midpoint, say mid, of the subarray, and consider the subarrays `A[low,...,mid]` and `A[mid+1,...,high]`.

Figure 15 shows, any contiguous subarray `A[i,..., j]` of `A[low,...,high]` must lie in exactly one of the following places:

- entirely in the subarray `A[low,...,mid]`, so that $low \le i \le j \le mid$,
- entirely in the subarray `A[mid+1,...,high]`, so that $mid < i \le j \le high$, or
- crossing the midpoint, so that $low \le i \le mid < j \le high$.

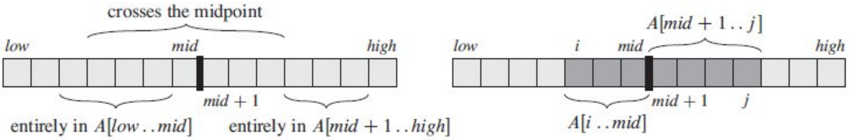


Figure: 15 Possible locations of maximum subarray of `A[low,...,high]`.

- We can find maximum subarrays of `A[low,...,mid]` and `A[mid+1,...,high]` recursively.

- Thus, all that is left to do is find a maximum subarray that crosses the midpoint, and take a subarray with the largest sum of the three.

- We can easily find a maximum subarray crossing the midpoint in time linear in the size of the subarray `A[low,...,high]`.

Algorithm 3 FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)

```
left_sum = −∞
sum = 0
for i = mid downto low do
    sum = sum + A[i]
    if sum > left_sum then
        left_sum = sum
        max − left = i
    end if
end for
right_sum = −∞
sum = 0
for j = mid + 1 to high do
    sum = sum + A[j]
    if sum > right_sum then
        left_sum = sum
        max_right = j
    end if
end for
return max_left, max_right, left_sum + right_sum
```

If the subarray `A[low,...,high]` contains n entries:

- $\Longrightarrow \qquad n =$ high - low $+ 1$
- We claim that the call `FIND-MAX-CROSSING-SUBARRAY(`A`, low, mid, high)` takes $\Theta(n)$ time.

Algorithm 4 FIND-MAXIMUM-SUBARRAY(A, low, high)

```
if high == low then
return (low, high, A[low])        //base case
else
    $mid = \lfloor (low + high)/2 \rfloor$
    (left-low, left-high, left-sum) = FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) = FIND-MAXIMUM-SUBARRAY(A, mid+1, high)
    (cross-low, cross-high, cross-sum) = FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
end if
if $left - sum \geq right - sum$ and $left - sum \geq cross - sum$ then return (left-low, left-high, left-sum)
else if $right - sum \geq left - sum$ and $right - sum \geq cross - sum$ then return (right-low, right-high, right-sum)
else
return (cross-low, cross-high, cross-sum)
end if
```

Analyzing the divide-and-conquer algorithm

We denote by $T(n)$ the running time of FIND-MAXIMUM-SUBARRAY on a subarray of n elements.

- Line 1 takes constant time.
- The base case, when $n = 1$, is easy: line 2 to takes constant time, and so
$$T(1) = \Theta(1). \tag{1}$$
- The recursive case occurs when $n > 1$. Assuming the array to be a power of 2, the subproblems solved in lines 5 and 6 is on a subarray of $n/2$ elements and takes a time of $T(n/2)$.
- As we have already seen, the call to FIND-MAX-CROSSING-SUBARRAY in line 7 takes $\Theta(n)$ time.
- Lines 9-13 take only $\Theta(1)$ time.

For the recursive case, therefore, we have

$$\begin{aligned} T(n) &= \Theta(1) + 2T(n/2) + \Theta(n) + \Theta(1) \\ &= 2T(n/2) + \Theta(n) \end{aligned} \qquad (2)$$

Combining equations (1) and (2) gives us a recurrence for the running time T(n) of FIND-MAXIMUM-SUBARRAY:

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1, \\ 2T(n/2) + \Theta(n), & \text{if } n > 1. \end{cases}$$

Matrix Multiplication

If $A = (a_{ij})$ and $B = (b_{ij})$ are square $n \times n$ matrices, then in the product $C = A \cdot B$, we define the entry c_{ij},

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}, \qquad \textit{for} \;\; i, j = 1, 2, ..., n$$

Algorithm 5 SQUARE-MATRIX-MULTIPLY(A, B)

```
n = A.rows
// let C be a new $n * n$ matrix
for $i = 1$ to $n$ do
    for $j = 1$ to $n$ do
        $c_{ij} = 0$
        for $k = 1$ to $n$ do
            $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
        end for
    end for
end for
return C
```

Because each of the triply-nested **for** loops runs exactly n iterations, the SQUARE-MATRIX-MULTIPLY procedure takes $\Theta(n^3)$ time.

A simple divide-and-conquer algorithm

Suppose that we partition each of A, B, and C into four $n/2 \times n/2$ matrices

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

so that we rewrite the equation $C = A \cdot B$ as

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix},$$

This equation corresponds to:

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21}, \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22}, \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21}, \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22}, \end{aligned}$$

A straightforward, recursive, divide-and-conquer algorithm:

Algorithm 6 SQUARE-MATRIX-MULT-RECUR(A, B)

```
$n = A.rows$
// let C be a new $n * n$ matrix
if n == 1 then
    $c_{11} = a_{11} \cdot b_{11}$
else
    $C_{11}$ = SQUARE-MATRIX-MULT-RECUR($A_{11}, B_{11}$) + SQUARE-MATRIX-MULT-RECUR($A_{12}, B_{21}$)
    $C_{12}$ = SQUARE-MATRIX-MULT-RECUR($A_{12}, B_{22}$) + SQUARE-MATRIX-MULT-RECUR($A_{12}, B_{22}$)
    $C_{21}$ = SQUARE-MATRIX-MULT-RECUR($A_{21}, B_{11}$) + SQUARE-MATRIX-MULT-RECUR($A_{22}, B_{21}$)
    $C_{22}$ = SQUARE-MATRIX-MULT-RECUR($A_{21}, B_{12}$) + SQUARE-MATRIX-MULT-RECUR($A_{22}, B_{22}$)
end if
return C
```

- Let $T(n)$ be the time to multiply two $n \times n$ matrices using this procedure.
- In the base case, when $n = 1$, we perform just the one scalar multiplication

  $T(1) = \Theta(1)$
- The recursive case occurs when $n > 1$.
  - We recursively call SQUARE-MATRIX-MULT-RECUR a total of eight times.
  - Each recursive call multiplies two $n/2 \times n/2$ matrices.
  - Contributing $T(n/2)$ to the overall running time,
  - The time taken by all eight recursive calls is $8T(n/2)$.

- Each of the matrices contains $n^2/4$ entries, and so each of the four matrix additions takes $\Theta(n^2)$ time.
- Since the number of matrix additions is a constant, the total time spent adding matrices is $\Theta(n^2)$.
- The total time for the recursive case, therefore is

$$\begin{aligned} T(n) &= \Theta(1) + 8T(n/2) + \Theta(n^2) \\ &= 8T(n/2) + \Theta(n^2) \end{aligned}$$

- Finally, the running time of SQUARE-MATRIX-MULT-RECUR:

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1, \\ 8T(n/2) + \Theta(n^2), & \text{if } n > 1. \end{cases}$$

The solution to this recurrence is $\Theta(n^3)$.

Strassen's method

The principal insight of Strassen's algorithm lies in the discovery that

- We can find the product C of two $n/2 \times n/2$ matrices, with just seven multiplications as opposed to the eight required by the brute-force algorithm.
- The cost of eliminating one matrix multiplication will be several new additions, but still only a constant number of additions.

Strassen's method follows the ff four steps:

1. Divide the input matrices A and B and output matrix C into $n/2 \times n/2$ submatrices as shown below.

   This step takes $\Theta(1)$ time by index calculation.

$$\left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right] = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] * \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right].$$

2. Create 10 matrices S_1, S_2,...,S_{10} , each of which is $n/2 \times n/2$ and is the sum or difference of two matrices created in step 1.

   We can create all 10 matrices in $\Theta(n^2)$ time.

3. Using the submatrices created in step 1 and the 10 matrices created in step 2, recursively compute seven matrix products P_1, P_2,..., P_7. Each matrix P_i is $n/2 \times n/2$.

4. Compute the desired submatrices C_{11}, C_{12}, C_{21}, C_{22} of the result matrix C by adding and subtracting various combinations of the P_i matrices.

   We can compute all four submatrices in $\Theta(n^2)$ time.

In step 2, we create the following 10 matrices:

$$S_1 = B_{12} - B_{22}, \quad S_2 = A_{11} + A_{12}, \quad S_3 = A_{21} + A_{22},$$
$$S_4 = B_{21} - B_{11}, \quad S_5 = A_{11} + A_{22}, \quad S_6 = B_{11} + B_{22},$$
$$S_7 = A_{12} - A_{22}, \quad S_8 = B_{21} + B_{22}, \quad S_9 = A_{11} - A_{21},$$
$$S_{10} = B_{11} + B_{12},$$

Since we must add or subtract $n/2 \times n/2$ matrices 10 times, this step does indeed take $\Theta(n^2)$ time.

In step 3, we recursively multiply $n/2 \times n/2$ matrices seven times to compute the following $n/2 \times n/2$ matrices, each of which is the sum or difference of products of A and B sub-matrices:

$$P_1 = A_{11} \cdot S_1 = A_{11} \cdot B_{12} - A_{11} \cdot B_{22},$$

$$P_2 = S_2 \cdot B_{22} = A_{11} \cdot B_{22} + A_{12} \cdot B_{22},$$

$$P_3 = S_3 \cdot B_{11} = A_{21} \cdot B_{11} + A_{22} \cdot B_{11},$$

$$P_4 = A_{22} \cdot S_4 = A_{22} \cdot B_{21} - A_{22} \cdot B_{11},$$

$$P_5 = S_5 \cdot S_6 = A_{11} \cdot B_{11} + A_{11} \cdot B_{22} + A_{22} \cdot B_{11} + A_{22} \cdot B_{22},$$

$$P_6 = S_7 \cdot S_8 = A_{12} \cdot B_{21} + A_{12} \cdot B_{22} - A_{22} \cdot B_{21} - A_{22} \cdot B_{22},$$

$$P_7 = S_9 \cdot S_{10} = A_{11} \cdot B_{11} + A_{11} \cdot B_{12} - A_{21} \cdot B_{11} - A_{21} \cdot B_{12}.$$

Step 4 adds and subtracts the P_i matrices created in step 3 to construct the four $n/2 \times n/2$ submatrices of the product C.

$$C_{11} = P_5 + P_4 - P_2 + P_6,$$

$$C_{12} = P_1 + P_2,$$

$$C_{21} = P_3 + P_4,$$

$$C_{22} = P_5 + P_1 - P_3 - P_7$$

When $n > 1$, steps 1, 2, and 4 take a total of $\Theta(n^2)$ time, and step 3 requires us to perform seven multiplications of $n/2 \times n/2$ matrices.

Hence, we obtain the following recurrence for the running time T(n) of Strassen's algorithm:

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 1, \\ 7T(n/2) + \Theta(n^2), & \text{if } n > 1. \end{cases}$$

This recurrence has a solution of $\Theta(n^{lg 7})$.

In conclusion, we have traded off one matrix multiplication for a constant number of matrix additions which actually leads to a lower asymptotic running time.

The substitution method for solving recurrences

The substitution method for solving recurrences comprises two steps:

1. Guess the form of the solution.
2. Use mathematical induction to find the constants and show that the solution works.

We can use the substitution method to establish either upper or lower bounds on a recurrence.

As an example, let us determine an upper bound on the recurrence $T(n) = 2T(\lfloor n/2 \rfloor) + n$.

- We guess that the solution is $T(n) = O(n\,lgn)$.
- The substitution method requires us to prove that $T(n) \leq cn\,lgn$ for an appropriate choice of the constant $c > 0$.

- We start by assuming that this bound holds for all positive $m < n$, in particular for $m = \lfloor n/2 \rfloor$, yielding $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor lg(\lfloor n/2 \rfloor)$.

- Substituting into the recurrence yields

$$\begin{aligned} T(n) &\leq 2c\lfloor n/2 \rfloor lg(\lfloor n/2 \rfloor) + n \\ &\leq cn\, lg(n/2) + n \\ &= cn\, lgn - cn\, lg2 + n \\ &= cn\, lgn - cn + n \\ &\leq cn\, lgn, \end{aligned}$$

where the last step holds as long as $c \geq 1$.

Mathematical induction now requires us to show that our solution holds for the boundary conditions.

- Then for $n = 1$, the bound $T(n) \leq cn\, lgn$ yields $T(n) \leq c * 1 * lg1 = 0$, which is at odds with $T(1) = 1$.

  Consequently, the base case of our inductive proof fails to hold.

- We can overcome this obstacle by taking an advantage of asymptotic notation requiring us only to prove $T(n) \leq cn\, lgn$ for $n \geq n_0$, where n_0 is a constant that we get to choose.

- Note that we can make a distinction between the base case of the recurrence ($n = 1$) and the base cases of the inductive proof ($n = 2$ and $n = 3$).

- With $T(1) = 1$, we derive from the recurrence that $T(2) = 4$ and $T(3) = 5$.

- Now we can complete the inductive proof that $T(n) \leq cnlgn$ for some constant $c \geq 1$ by choosing c large enough so that $T(2) \leq c2lg2$ and $T(3) \leq c3lg3$.

- As it turns out, any choice of $c \geq 2$ suffices for the base cases of $n = 2$ and $n = 3$ to hold.

The recursion-tree method for solving recurrences

- In a recursion tree, each node represents the cost of a single subproblem somewhere in the set of recursive function invocations.

- We sum the costs within each level of the tree to obtain a set of per-level costs, and then we sum all the per-level costs to determine the total cost of all levels of the recursion.

- A recursion tree is best used to generate a good guess, which you can then verify by the substitution method.

For example: let us see how a recursion tree would provide a good guess for the recurrence $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$.

- Start by focusing on finding an upper bound for the solution.
- Create a recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$, having written out the implied constant coefficient $c > 0$.
- Figure 16 shows how we derive the recursion tree by assuming for convenience that n is an exact power of 4.

The recursion-tree method for solving recurrences

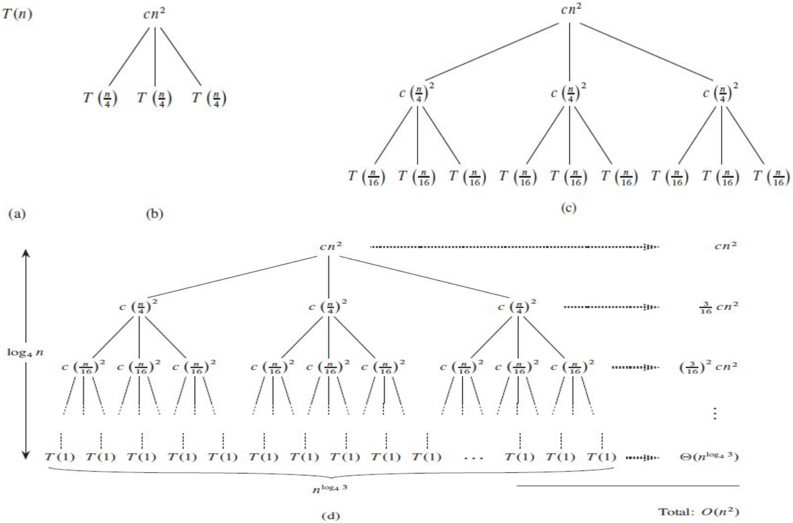

$T(n)$
cn^2
$T\left(\frac{n}{4}\right)$
$c\left(\frac{n}{4}\right)^2$
$T\left(\frac{n}{16}\right)$
(a)
(b)
(c)
$\log_4 n$
$c\left(\frac{n}{16}\right)^2$
$T(1)$
cn^2
$\frac{3}{16}cn^2$
$\left(\frac{3}{16}\right)^2 cn^2$
$\Theta(n^{\log_4 3})$
$n^{\log_4 3}$
(d)
Total: $O(n^2)$

Figure: 16 Constructing a recursion tree for the recurrence $T(n) = 3T(n/4) + cn^2$. Part (a) shows $T(n)$, which progressively expands in (b)-(d) to form the recursion tree. The fully expanded tree in part (d) has height $\log_4 n$ (it has $\log_4 n + 1$ levels).

Because subproblem sizes decrease by a factor of 4 each time we go down one level, we eventually must reach a boundary condition.

- The subproblem size for a node at depth i is $n/4^i$.
- Thus, the subproblem size hits $n = 1$ when $n/4^i = 1$ or, equivalently, when $i = \log_4 n$.
- Thus, the tree has $\log_4 n + 1$ levels (at depths 0, 1, 2,...,$\log_4 n$).

Next we determine the cost at each level of the tree.

- Each level has three times more nodes than the level above, and so the number of nodes at depth i is 3^i.
- A node at depth i, has a cost of $c(n/4^i)^2$, and the total cost will be $3^i c(n/4^i)^2 = (3/16)^i cn^2$.
- The bottom level, at depth $\log_4 n$, has $3^{\log_4 n} = n^{\log_4 3}$ nodes, each contributing cost $T(1)$, for a total cost of $n^{\log_4 3} * T(1)$, which is $\Theta(n^{\log_4 3})$.

Now we add up the costs over all levels to determine the cost for the entire tree:

$$
\begin{aligned}
T(n) &= cn^2 + \frac{3}{16}cn^2 + (\frac{3}{16})^2 cn^2 + ... + (\frac{3}{16})^{\log_4 n - 1} cn^2 + \Theta(n^{\log_4 3}) \\
&= \sum_{i=0}^{\log_4 n - 1} (\frac{3}{16})^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{(\frac{3}{16})^{\log_4 n} - 1}{(\frac{3}{16}) - 1} cn^2 + \Theta(n^{\log_4 3})
\end{aligned}
$$

$$
\begin{aligned}
T(n) &= \sum_{i=0}^{\log_4 n-1} (\frac{3}{16})^i cn^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} (\frac{3}{16})^i cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{1}{1-(\frac{3}{16})} cn^2 + \Theta(n^{\log_4 3}) \\
&= \frac{16}{13} cn^2 + \Theta(n^{\log_4 3}) \\
&= O(n^2)
\end{aligned}
$$

- Thus, we have derived a guess of $T(n) = O(n^2)$ for our original recurrence $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$.
- Therefore, the cost of the root dominates the total cost of the tree.

Now we can use the substitution method to verify that our guess was correct.

- We want to show that $T(n) \le dn^2$ for some constant $d > 0$.
- Using the same constant $c > 0$ as before, we have

$$\begin{aligned} T(n) &\le 3T(\lfloor n/4 \rfloor) + cn^2 \\ &\le 3d\lfloor n/4 \rfloor^2 + cn^2 \\ &\le 3d(n/4)^2 + cn^2 \\ &= \frac{3}{16}dn^2 + cn^2 \\ &\le dn^2 \end{aligned}$$

where the last step holds as long as $d \ge (16/13)c$.

The master method for solving recurrences

The master method for solving recurrences depends on the following theorem.

Theorem 1.2 (Master theorem)

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be asymptotically positive 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 $T(n)$ has the following asymptotic bounds:

- If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
- If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} lgn)$.
- 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 $T(n) = \Theta(f(n))$.

In each of the three cases, we compare the function $f(n)$ with the function $n^{\log_b a}$.

- In case 1 the function $n^{\log_b a}$ is larger while in case 3 its the vice versa.
- In case 2, the two functions are of the same size.

Some additional technicalities that needs to be considered when using the master method are:

- In the 1^{st} case, not only must $f(n)$ be smaller than $n^{\log_b a}$, it must be polynomially smaller.
  - That is, $f(n)$ must be asymptotically smaller than $n^{\log_b a}$ by a factor of n^{ϵ} for some constant $\epsilon > 0$.
- In the 3^{rd} case, not only must $f(n)$ be larger than $n^{\log_b a}$, it also must be polynomially larger and in addition satisfy the "regularity" condition that $af(n/b) \leq cf(n)$.

Note that the three cases do not cover all the possibilities for $f(n)$.

Examples:

1. Consider $T(n) = 9T(n/3) + n$
   - For this recurrence, we have $a = 9$, $b = 3$, $f(n) = n$.
   - 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 $T(n) = \Theta(n^2)$.

2. Consider $T(n) = T(2n/3) + 1$

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

3. Consider $T(n) = 3T(n/4) + nlgn$

   - For this recurrence $a = 3$, $b = 4$, $f(n) = nlgn$, 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, we have that

$$af(n/b) = 3f(n/4) \leq 3/4nlgn = cf(n) \text{ for } c = 3/4.$$

Consequently, by case 3, the solution to the recurrence is $T(n) = \Theta(n\,lgn)$.

The hiring problem

Suppose that you need to hire a new office assistant using an employment agency.

- The employment agency sends you one candidate each day.
- You interview that person and then decide either to hire that person or not.
- You must pay the employment agency a small fee to interview an applicant.
- To actually hire an applicant is more costly, however, since you must fire your current office assistant and pay a substantial hiring fee to the employment agency.
- You are committed to having, at all times, the best possible person for the job.

The procedure HIRE-ASSISTANT, given below, expresses this strategy for hiring in pseudocode.

Algorithm 7 HIRE-ASSISTANT(n)

```
$Best = 0$    //candidate 0 is a least-qualified dummy candidate
for $i = 1$ to $n$ do
    interview candidate i
    if candidate $i$ is better than candidate $best$ then
        $best = i$
        hire candidate $i$
    end if
end for
```

The cost model of HIRE-ASSISTANT is dependant on

- The costs incurred by interviewing and hiring not on the running time.
- Interviewing has a low cost, say c_i, whereas hiring is expensive, costing c_h.
- Letting m be the number of people hired, the total cost associated with this algorithm is $O(c_i n + c_h m)$.

Worst-case analysis

- In the worst case, we actually hire every candidate that we interview.
- As a result we hire n times, for a total hiring cost of $O(c_h n)$.

Probabilistic analysis

- Probabilistic analysis is the use of probability in the analysis of problems.

- In order to perform a probabilistic analysis on hiring procedure, we must use knowledge of, or make assumptions about, the distribution of the inputs.

- Then the output is an average-case running time, where we take the average over the distribution of the possible inputs.

For the hiring problem, we can assume that the applicants come in a random order.

- Thus, we can rank each candidate with a unique number from 1 through n, using `rank(i)` to denote the rank of applicant i.
- Random order $\iff$ ranks form a uniform random permutation; that is, each of the possible $n!$ permutations appears with equal probability.

Randomized algorithms

In order to use probabilistic analysis, we need to know something about the distribution of the inputs.

- We call an algorithm randomized if its behavior is determined not only by its input but also by values produced by a random-number generator (`RANDOM`).
- A call to `RANDOM(a, b)` returns an integer between a and b, inclusive, with each such integer being equally likely.

- When analyzing the running time of a randomized algorithm, we take the expectation of the running time over the distribution of values returned by the random number generator.
- In general,
  - average-case running time ⟹ probability distribution is over the inputs to the algorithm, and
  - expected running time ⟹ the algorithm itself makes random choices.

Indicator random variables

Indicator random variables provide a convenient method for converting between probabilities and expectations.

- Suppose we are given a *sample space* S and an *event* A. Then the indicator random variable $I\{A\}$ associated with event A is defined as:

$$I\{A\} = \begin{cases} 1 & \text{if A occurs,} \\ 0 & \text{if A does not occur.} \end{cases}$$

Example:

let us determine the expected number of heads that we will obtain when flipping a fair coin.

$S = \{H, T\}$, with $Pr\{H\} = Pr\{T\} = 1/2$.

$$X_H = I\{H\} = \begin{cases} 1 & \text{if H occurs,} \\ 0 & \text{if T occurs.} \end{cases}$$

The expected number of heads obtained in one flip is:

$$\begin{aligned} E[X_H] &= 1 \cdot Pr\{H\} + 0 \cdot Pr\{T\} \\ &= 1 \cdot (1/2) + 0 \cdot (1/2) \\ &= 1/2. \end{aligned}$$

What is the expected number of heads for n fair coin flips?

- Let $X_i = I\{\text{the } i\text{th flip results in the event } H\}$.
- Let X be the random variable denoting the total number of heads in the n coin flips, so that

  $X = \sum_{i=1}^{n} X_i, \quad \Rightarrow$ the expectation of the sum of n indicator random variables.

Finally, we can easily compute the expected number of heads:

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^{n} X_i\right] \\ &= \sum_{i=1}^{n} E[X_i] \quad \Rightarrow \quad \textit{linearity of expectation} \\ &= \sum_{i=1}^{n} 1/2 \\ &= n/2. \end{aligned}$$

Analysis of the hiring problem using indicator random variables

Let X be the random variable whose value equals the number of times we hire a new office assistant.

$$E[X] = \sum_{i=1}^{n} x \, Pr\{X = x\}.$$

Let X_i be the indicator random variable associated with the event in which the i^{th} candidate is hired. Thus,

$$X_i = \begin{cases} 1 & \text{if candidate i is hired,} \\ 0 & \text{if candidate i is not hired.} \end{cases}$$

and
$X = X_1 + X_2 + ... + X_n \Rightarrow$ gives us the number of times we hire a new office assistant.

- $E[X_i] = Pr\{\text{candidate } i \text{ is hired}\}$, which occurs when candidate i is better than each of candidates 1 through $i-1$.

- Since, all the candidates are equally likely of being the best-qualified so far $\quad\Rightarrow\quad E[X_i] = 1/i$.

Now we can compute $E[X]$:

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^{n} X_i\right] \\ &= \sum_{i=1}^{n} E[X_i] \\ &= \sum_{i=1}^{n} 1/i \\ &= \ln n + O(1). \end{aligned}$$

Therefore, even though we interview n people, we actually hire only approximately $\ln n$ of them, on average.

Randomized algorithms

In probabilistic analysis, the algorithm is deterministic; for any particular input, the number of times a new office assistant is hired is always the same.

- Given the rank list $A_1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$, a new office assistant is always hired 10 times.
- Given the list of ranks $A_2 = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]$, a new office assistant is hired only once, in the first iteration.
- Given a list of ranks $A_3 = [5, 2, 1, 8, 4, 7, 10, 9, 3, 6]$, a new office assistant is hired three times.

Consider, a randomized algorithm that first permutes the candidates and then determines the best candidate.

- In this case, we randomize in the algorithm, not in the input distribution.

Given a particular input, say A_3 above,

- We cannot say how many times the maximum is updated, because this quantity differs with each run of the algorithm.
- For this algorithm and many other randomized algorithms, no particular input elicits its worst-case behavior.

For the hiring problem, the only change needed in the code is to randomly permute the array.

Algorithm 8 RANDOMIZED-HIRE-ASSISTANT(n)

```
randomly permute the list of candidates
$Best = 0$    //candidate 0 is a least-qualified dummy candidate
for $i = 1$ to $n$ do
    interview candidate i
    if candidate $i$ is better than candidate $best$ then
        $best = i$
        hire candidate $i$
    end if
end for
```

- The expected hiring cost of the procedure `RANDOMIZED-HIRE-ASSISTANT` is $O(c_h \ln n)$.
- In probabilistic analysis, we make an assumption about the input.
- In randomized algorithms we make no such assumption, although randomizing the input takes some additional time.

End of Chapter 1

Questions?