3. Algorithm analysis and brute force

1. Algorithm analysis and Big-O notation
2. Recurrence relations for time functions
3. Brute force and nested loops
4. Exhaustive search

Inquiry

• What is algorithm performance?
• Is time scalability a good way to measure the performance of computing systems?
• How much does studying design approaches help us create efficient solutions?
• How hard is optimization?
Guess

What are the running times for some solutions to:

– Finding duplicates
– Sorting an array
– Finding closest-pair of \((x, y)\) points
– Satisfiability of logic formulas
– Towers of Hanoi (Brahma)

Topic objective

Use recurrences and asymptotic complexity notation to analyze array-traversal and other brute-force algorithms.
Background objectives

0.3a State and explain the running time of an array traversal, as a function of array size**
0.3b Explain or apply a concept in set theory**
0.3c Describe a function*

See Background slides and study questions for support with these

1. Algorithm analysis and Big-O notation

• What is an upper or lower time bound?
• What upper time bound does every algorithm have?
• Can you arrange these in order of how fast the function graph rises?

\[ f(x) = x^2 \quad f(x) = \log_2(x) \]
\[ f(x) = 8 \quad f(x) = 2^x \quad f(x) = x/2 \]
Subtopic objectives

3.1 Define or apply the big-O, θ, and big-Ω notations**

Exercise

• Arrange these in order of how fast the value returned by the function rises
  
a. \( f(n) = 6 \lg n \)
  
b. \( f(n) = 5 n^3 \)
  
c. \( f(n) = n^2/8 + 100n \)
  
d. \( f(n) = 2^n \)
  
e. \( f(n) = 10 n \lg n + 100 \)
  
f. \( f(n) = 80n + 1000 \)
Wanted: a way to measure performance

- ..that is independent of
  - hardware
  - size of input
  - contents of input
  - programming language
- … and is mathematically precise

A solution

- We seek to describe running time as a function of the size of input
- Example: The running time of an array of traversal is proportional to (i.e., a linear function of) the size of the array
Assumptions about hardware

We assume that

- steps involving single machine instructions, e.g., assignment, arithmetic operations, comparison, take one time unit each
- architecture is random-access machine, i.e., variable access is constant-time, including for array elements
- I/O steps take constant time

Best, worst, and average cases

- Best or worst case specifies efficiency with input data that yields lowest or highest running time

- Example:
  - Best case for linear search is when $key = A[1]$
  - Worst case is key in last position

- Average case (expected running time) is not necessarily the average of best and worst cases
Empirical analysis of algorithms

- *Time* algorithm or *count steps*, for different quantities of input
- Plot time vs. data quantity as scatterplot
- Interpolate/extrapolate to derive time function
- *Exercise*: Plot running time of Quicksort for data sizes of 100, 200, 400, 800, 1600, 3200
- Our emphasis is *theoretical* analysis (whose usefulness is tested empirically)

Big-O notation

- Quantifies and classifies the growth rates of functions
- Big-O notation may express the time efficiency of an algorithm as a function of amount of data processed
- We abstract from details and simplify to obtain platform-independent results
Ignoring constants

• One algorithm is “asymptotically faster” than another if it is faster for all sizes of inputs, within a constant factor, beyond some constant additive term

• Examples:
  – \( O(n + 1,000,000) = O(n) \)
  – \( O(0.2 \lg n) = O(\lg n) \)
  – \( O(k n^2) = O(n^2) \) for any constant \( k \)
  – \( O(n^2 + 5n) = O(n^2) \), \( O(n + \log n) = O(n) \)

Loops and complexity

• A single loop of \( n \) iterations, where each iteration executes \( O(1) \) steps, is \( O(n) \)

• A loop nested to two levels, each with roughly \( n \) iterations, where each iteration executes \( O(1) \) steps, is \( O(n^2) \)

• If we start with \( n \) items to look at and cut our remaining work in half at each step, then the job will take \( O(\log_2 n) \) such steps.

• If our loops are nested to \( n \) levels, as in password guessing, then algorithm is \( O(2^n) \) — offer job to someone else
3. Algorithm analysis; brute force

### Decision trees

- Diagram a branching algorithmic process
- Root is starting point; leaves are terminations
- Each branching step is a node
- Edges reflect alternative choices
- Depth of tree expresses algorithm’s running time

![Decision Tree Diagram]

### Decision trees and running time

- Execution of an algorithm may be represented as descending a *decision tree*, from root to leaf
- **Examples:**
  - Binary search [Johnsonbaugh, p. 168]
  - Sort [p. 255]
- **Theorems based on decision trees:**
  - Worst-case time for comparison-based search of sorted array is $\Omega(lg \, n)$, hence binary search is asymptotically optimal
  - Worst-case time for comparison-based sorting is $\Omega(n \, lg \, n)$
Asymptotic behavior

- Different time functions have graphs that “cross over” each other
- For amounts of data up to $k$, time function $f$ is “better,” so its algorithm is more efficient
- But above $k$, $g$ shows less time
- We choose the algorithm described by $g$, because eventually (asymptotically) its time value $T$ will be less than $f$’s
  - …because $g$ grows more slowly than $f$

Upper and lower bounds

- Function $g$ is an *upper bound* on function $f$ iff $(\forall x \in \mathbb{N}) g(x) \geq f(x)$
- Suppose
  - $f(x) = x^2$
  - $g(x) = x^3$
  - $h(x) = (x - 1) / 2$
- Then
  - $g$ is an upper bound on $f$
  - $f$ is a lower bound on $g$
  - $h$ is a lower bound on $f$ and $g$
(See graphs of the functions)
### Big-O notation

- \( f \in O(g) \) iff
  \[
  \exists c, k \quad (\forall x \geq k) \quad 0 \leq f(x) \leq cg(x)
  \]
- Hence \( O(g) \) is the set of functions for which \( g \) is an upper bound within a constant factor
- The graph of \( cg(n) \) rises above the graph of \( f(n) \) everywhere for \( n > k \) (never cross again)

### Grouping time functions

- Functions that grow roughly at the same rate are in the same Big-O category
- Constant time: \( O(1) \)
- Logarithmic time: \( O(\log_2 n) = O(\lg n) \)
- Linear time: \( O(n) \)
- \( N \)-log: \( O(n \lg n) \)
- Quadratic time: \( O(n^2) \)
- Exponential time: \( O(2^n) \)
Dominance relations on functions

• Our goal is to obtain a hierarchy of sets of time functions to classify algorithm runtimes
• $O(n)$ functions dominate $O(lg\ n)$, $O(n^2)$ dominates $O(n)$, etc.
• Non-dominant terms in time functions are discarded
• Constant terms and factors are discarded
• Example: $O(3n^2 + 2n – 100)$ simplifies to $O(n^2)$

Big-O expressions that dominate others

• Theorem:
  $(\forall n) \ T_1(n) \in O(g_1(n)) \land T_2(n) \in O(g_2(n))
  \implies T_1(n) + T_2(n) \in O(\max\{g_1(n), g_2(n)\})$

• Discussion: The times of two steps of an algorithm, added together, grow as the order of the slower (maximum-time) part
• $O(n)$ functions dominate $O(lg\ n)$, $O(n^2)$ dominates $O(n)$, etc.
3. Algorithm analysis; brute force

Slower part of an algorithm dominates its running time

- **Theorem:**
  \[(\forall n) \, T_1(n) \in O(g_1(n)) \land T_2(n) \in O(g_2(n)) \implies T_1(n) + T_2(n) \in O(\max\{g_1(n), g_2(n)\})\]

- **Discussion:** The times of two steps of an algorithm, added together, grow as the order of the slower (maximum-time) part

- **Example:** Algorithm that checks for duplicates in an array by bubble-sorting then checking consecutive elements, is $O(n^2)$

Big-omega notation ($\Omega$)

- Expresses *lower bound* on time for optimistic or best case

- **Example:** A Find-max algorithm for arrays is $\Omega(n)$ because any algorithm must check every array element

- Precise characterization of comparison-based sorting: $\Omega(n \lg n)$

- A problem’s complexity may be expressed using $\Omega$ if the problem is known to have a lower bound on time
Big-omega and lower bound

- \( f \in \Omega(g) \) iff
  \( (\exists \ c, \ n_0) (\forall x \geq n_0) \ 0 \leq cg(x) \leq f(x) \)
- [pic] the graph of \( f(n) \) rises above the graph of \( cg(n) \) everywhere for \( n > n_0 \) (never cross again)
- \( \Omega \) characterizes problems; e.g., searching unordered array is \( \Omega(n) \); sorting by comparisons is \( \Omega(lg \ n) \)

Theta notation (\( \Theta \))

- \( \Theta \) expresses a tight bound (upper and lower) on the complexity of an algorithm
- Definition: an algorithm is \( \Theta(f(n)) \) iff it is \( O(f(n)) \) and also \( \Omega(f(n)) \)
- Functions that are \( \Theta(f(n)) \) do not rise much slower or faster than \( f(n) \)
- Examples:
  - An algorithm that is \( O(n) \) is also \( O(n^2) \)
  - One that is \( \Theta(n) \) is too fast to be \( \Theta(n^2) \)
  - One that is \( \Theta(n^2) \) is too slow to be \( \Theta(n) \)
### Formal definitions of $O$, $\Omega$, $\Theta$

- These define **sets of functions that increase at a similar rate**
- $f \in O(g)$ iff 
  \[ (\exists \ c, \ k) \ (\forall x) \ f(x) \leq cg(x) + k \]
- $f(x) \in \Omega (g(x))$ iff 
  \[ \exists \ c, \ n_0 \ \text{s.t.} \ f(x) \geq cg(x) + n_0 \]
- $f(x) \in \Theta(g(x))$ iff 
  \[ f(x) \in O(g(x)) \land f(x) \in \Omega (g(x)) \]

### Notation conventions

**Functions:**
- We often write $f(n)$ for $f : \mathbb{N} \rightarrow \mathbb{N}$
- $f(n) \in \theta(n^d)$ means “$f : \mathbb{N} \rightarrow \mathbb{N}$ is in efficiency class $\theta(g(n))$, where $g(n) = n^d$, i.e., $g$ is a $d$-degree polynomial”
- **Example:** if $f(n) = 5n^2 - 80$, then $f(n) \in \theta(n^2)$

**Arrays:**
- $A[1..|A|-1]$ means the sequence in $A$ from the first to the next-to-last
- $A + \langle x \rangle$ means array $A$ concatenated with the array consisting of the element value $x$
Algorithm analysis

- **Analysis**: separation into components
- Form of running-time analysis for algorithm $\alpha$, with running time that is big-$O$ or big-$\Theta$ (theta) of $f$, where $f$ is a function of the size of $\alpha$’s input:
  \[ T_\alpha(n) = O(f(n)) \]
- **Examples**:
  - $T_{\text{Array-append}}(n) = \Theta(1)$
  - $T_{\text{Linear-search}}(n) = \Theta(n)$
  … because appending a new element to an array takes constant time and linear search takes time proportional to the array size

O(), $\Omega()$, $\Theta()$, and algorithms

- O(), $\Omega()$, $\Theta()$ expressions denote *sets of functions*
- In algorithm analysis, this is used with *time functions for algorithms*
- So, for example, $\Theta(n)$ is not “slower” than $\Theta(\lg n)$, even though algorithms with *time function* in $\Theta(n)$ are slower than those with time functions in $\Theta(\lg n)$
Comparing rates of growth

- Let \( q = \lim_{n \to \infty} \frac{T(n)}{g(n)} \)
- If \( q = 0 \) then \( T(n) \in O(g(n)) \)
- If \( q \) is constant >0 then \( T(n) \in \Theta(g(n)) \)
- If \( q = \infty \) then \( T(n) \in \Omega(g(n)) \)
- *Example*: let \( T(n) = \log_2 n \), \( g(n) = \sqrt{n} \);
  \[
  \lim_{n \to \infty} \log_2 n = 2 \log_2 e \lim_{n \to \infty} \frac{\sqrt{n}}{n} = 0;
  \]
  hence \( \log_2 n \in O(\sqrt{n}) \)

2. Recurrence relations for time functions

*Exercises:*
- What is analysis?
- Write a recurrence for \( f(n) = 2^n \)
3. Algorithm analysis; brute force

Subtopic objective

3.2 Write and solve a recurrence for the time function of a linear-time algorithm**

Recurrences and time analysis

- Recurrences are function definitions
- Their form is different from pseudocode
  - There are curly braces in recurrences
  - In recurrences, the return value goes on the left and the IF goes on the right.
- In running-time recurrences, the solution is a big-O or Θ expression.
Example problem: array search

1. Iterative version:

   \[ \text{Search}(A, \text{key}) \]
   \[
i \leftarrow 1
   \]
   \[
   \text{While } i \leq |A|
   \]
   \[
   \quad \text{if } A[i] = \text{key return true}
   \]
   \[
   \quad i \leftarrow i + 1
   \]
   \[
   \text{Return false}
   \]

2. Recursive definition:

   \[ \text{Search}(A, \text{key}) = \]
   \[
   \begin{cases} 
   \text{False} & \text{if } |A| = 0 \\
   \text{True} & \text{if } A[1] = \text{key} \\
   \text{Search}(A[2 .. |A|], \text{key}) & \text{otherwise}
   \end{cases}
   \]

3. Time recurrence

   – It takes one step to test size of array,
   – plus one to compare \( A[1] \) to \( \text{key} \),
   – plus the number of steps to search the rest of the array, starting at 2

   \[
   T_{\text{Search}}(n) = \begin{cases} 
   3 & \text{if } n \leq 1 \\
   3 + T_{\text{Search}}(n - 1) & \text{otherwise}
   \end{cases}
   \]

4. Solution to time recurrence: \( O(n) \)

   because time for base case is constant (\( O(1) \)) and depth of recursion is \( (n - 1) \), i.e., \( O(n) \) and \( O(n) \times O(1) = O(n) \)
Algorithm analysis using recurrences

1. Define algorithm iteratively in pseudocode
2. Define recurrence relation that specifies function computed by algorithm
3. Based on (2), define recurrence that relates running time of algorithm to size of input
4. Solve recurrence relation in \( O, \Omega, \theta \) notation, using the notion of depth of recursion (number of times algorithm invokes self)

Running time and depth of recursion

- When a method calls itself, the time required is the sum of the time of the base step plus the time of the recursive step
- Time of recursive step has the same property
- Hence total time required is \( product \) of base-step time and depth of the recursive calls
- \textit{Example:} Time of factorial\((n)\) is \( O(1) \times O(n) \); i.e. time to multiply times depth of recursion \((n - 1)\)
## Decrease by one

Decrease-by-one algorithms reduce the remaining data to be processed by one at each step

### Factorial \((n)\)

\[
\text{if } n \leq 1 \\
\quad \text{return } 1 \\
\text{else} \\
\quad \text{return } n \times \text{Factorial}(n - 1)
\]

**Analysis:** Factorial is \(O(n)\)

---

## Solving decrease-by-one recurrences

- Decrease-by-one algorithms exploit the relationship between \(f(n)\) and \(f(n-1)\), e.g., \(n! = n(n-1)!\)
- These algorithms’ time efficiency may be expressed in the form \(T(n) = T(n-1) + f(n)\) where \(f(n)\) expresses time to reduce and extend between smaller and larger instances

\[
T(n) = T(n - 1) + f(n) \\
= T(n - 2) + f(n - 1) + f(n) \ldots
\]
3. Algorithm analysis; brute force

**Linear-time examples**

- Maximum element of array
- Sum of array elements
- Insertion into sorted array
- Index of minimum array element
- Traversal of linked list
- Array partition
- Merge arrays

---

**Performance of max algorithm**

(a) *Set definition:* \( \text{max}(A) = x \in A \text{ s.t. } (\forall i \leq |A|) \ x \geq A[i] \)

(b) *Algorithmic definition* for max element of array:

\[
\text{Max-elt}(A) = \begin{cases} 
\uparrow & \text{if } |A| = 0 \\
A[1] & \text{if } |A| = 1 \\
\max\{A[1], \text{Max-elt}(A[2 \ldots |A|])\} & \text{otherwise}
\end{cases}
\]

\[
T_{\text{Max-elt}}(n) = \begin{cases} 
O(1) & \text{if } n = 0, 1 \\
O(1) + T_{\text{Max-elt}}(n - 1) & \text{otherwise}
\end{cases}
\]
Adding elements of an array

**Array-sum(A)**

\[
\begin{align*}
i &\leftarrow 1 \\
y &\leftarrow 0 \\
\text{while } i \leq |A| &\leftarrow \\
&\quad y \leftarrow y + A[i] \\
&\quad i \leftarrow i + 1 \\
\text{return } y &\leftarrow \\
\end{align*}
\]

Analysis of array-sum

**Recurrence for Array-sum:**

\[
\begin{align*}
\text{Sum}(A) &= \\
\{ 0 &\quad \text{if } |A| = 0 \\
A[1] &\quad \text{if } |A| = 1 \\
A[1] + \text{Sum}(A[2 .. |A|]) &\quad \text{otherwise}
\end{align*}
\]

**Time recurrence:**

\[
\begin{align*}
T_{\text{Sum}}(n) &= \\
\{ O(1) &\quad \text{if } n \leq 1 \\
O(1) + T_{\text{Sum}}(n-1) &\quad \text{otherwise}
\end{align*}
\]

= \( O(n) \)
3. Algorithm analysis; brute force

Array insertion
(used by Insertion Sort)

**Insert-ascending** \((A, x)\)

\[
i \leftarrow |A|
\]

while \(i > 1\) and \(x < A[i]\)

\[
A[i] \leftarrow A[i - 1]
\]

\[
i \leftarrow i - 1
\]

\[
A[i] \leftarrow x
\]

Return \(A\)

- Size of input initializes counter
- Loop iterates up to \(n\) times
- \(T_{\text{Insert-ascending}}(n) = O(n)\)

Recursive array insertion

**Insert-ascending** \((A, x)\)

If \(|A| = 0\)

return \(\langle x \rangle\)

else

if \(x < A[1]\)

return \(\langle x \rangle + A\)

else

return \(\langle A[1] \rangle +

Insert-ascending(A[2 .. |A|], x)\)
3. Algorithm analysis; brute force

**Insert-ascend (A, x)** =
\[
\begin{cases}
\langle x \rangle & \text{if } |A| = 0 \\
\langle x \rangle + A & x < A[1] \\
\langle A[1] \rangle + \text{Insert-ascend}(A[2..|A|], x) & \text{otherwise}
\end{cases}
\]

\[
T_{\text{insert-ascend}}(n) =
\begin{cases}
1 & \text{if } n \leq 1 \\
O(1) + T_{\text{insert-asc}}(n - 1) & \text{otherwise}
\end{cases}
= O(n)
\]

**Min-index**

- Used by *Selection-sort*
- Returns index of smallest element of array A

\[
\text{Min-index}(A) =
\begin{cases}
1 & \text{if } |A| = 1 \\
\min\{A[1], A[\text{min-index}(A[2..|A|])]\} & \text{otherwise}
\end{cases}
\]
### Recursive linked-list traversal

**Traverse** *(Node-ptr)*

If *Node-ptr* not null

Visit node pointed to by *Node-ptr*

*Traverse* *(Next (Node-ptr))*

- **Complexity:** \( T(n) = \begin{cases} 
1 & \text{for null list} \\
2 + T(n - 1) & \text{otherwise} 
\end{cases} \)

- **Solution:** \( O(n) \) is running time of *Traverse*, assuming constant time of visiting one node and assuming constant time of *next* method

### Intuition for array partition

- Used in Quicksort
- Partition algorithm copies all elements of \( A \) less than pivot into array \( B \), all others to array \( C \), and copies \( B \), pivot, and \( C \) back into \( A \)
- Partition works by repeatedly finding leftmost element in left partition that should be in right and rightmost that should be in left, and swapping them
3. Algorithm analysis; brute force

**Partition(A)**

\[ pv \leftarrow A[1] \]
\[ j \leftarrow 1 \]
\[ k \leftarrow 1 \]

for \( i \leftarrow 2 \) to \(|A|\)

if \( A[i] < pv \)

\[ B[j] \leftarrow A[i] \]
\[ j \leftarrow j + 1 \]

else

\[ C[k] \leftarrow A[i] \]
\[ k \leftarrow k + 1 \]

\[ i \leftarrow i + 1 \]

\[ A \leftarrow B[1..j-1] + pv + C[1..k-1] \]

return \((A, j)\)

---

**Merging two sorted arrays**

*Destination array*

\[ C \]
\[ Ann \quad Bill \quad \]

*Source arrays*

\[ A \]
\[ Bill \quad Charli \quad Dan \quad ... \]

\[ B \]
\[ Ann \quad Cliff \quad Eva \quad ... \]

Repeat until A or B is exhausted:

\[ x \leftarrow \min \{A[i], B[bi]\} \]

append \( x \) to the end of \( C \), incrementing \( ci \)

increment \( ai \) or \( bi \) as appropriate

Copy remainder of A or B to end of C

- \( C \) should be as large as A, B, together
3. Algorithm analysis; brute force

**Merge running time**

A and B are sorted arrays; Merge (A, B) returns a sorted array containing all elements of A, B.

\[ \text{Merge} (A, B) = \begin{cases} 
A & \text{if } B = \lambda \\
B & \text{if } A = \lambda \\
B[1] + \text{Merge}(A, B[2..|B|]) & \text{otherwise}
\end{cases} \]

*Spec:* If A, B are each sorted arrays, then \( \text{Merge}(A, B) \) is a sorted array.

*Problem:* Write \( T_{\text{Merge}}(n) \) recurrence

---

**Linear time functions**

\[
T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
1 + T(n - 1) & \text{otherwise}
\end{cases}
\]

*Solution:* \( \Theta(n) \), because one step is needed for base case at each level of recursion, and there are \( (n - 1) \) levels.

*Examples:* Evaluation of each function on previous slide takes one step for base case and depth of recursion is \( \Theta(n) \)
3. Brute force and nested loops

- What is the most straightforward way to design an algorithm?
- What are the loops like in the sorting algorithms you know?

Subtopic objectives

3.3a Explain the brute-force approach to algorithm design*
3.3b Describe the running time of a nested-loop algorithm*
3.3c Write and solve a time recurrence for a nested-loop algorithm*
### Brute-force design

- Brute-force algorithms follow straightforwardly from their problem statements
- **Advantages:**
  - applicable to a wide range of problems
  - simple to design
  - may be useful to solve small instances
- **Disadvantage:** inefficient in general case

### Finding duplicates

**Has-dups(A)**

For \( i \leftarrow 1 \) to \(|A| - 1\)

for \( j \leftarrow 2 \) to \(|A|\)


return true

Return false

\[
\text{has-dups}(A) = \begin{cases} 
  \text{false} & \text{if } |A| < 2 \\
  \text{true} & \text{if search}(A[2..|A|], A[1]) \\
  \text{has-dups}(A[2 .. |A|]) & \text{otherwise}
\end{cases}
\]
Complexity of nested loops

- Running time of the algorithm is the product of the running times of the nested loops
- Example: Has-dups (previous slide) has running time of $O(n)O(n) = O(n^2)$

String search

$\text{Search}(S[1..n], \text{key}[1..m])$

> Returns location of first occurrence of $\text{key}$ in $S$

for $i \leftarrow 1$ to $n - m + 1$ do
  $j \leftarrow 1$
  while $j \leq m \land \text{key}[j] = S[i + j]$
    $j \leftarrow j + 1$
  if $j > m$ return $i$
return $-1$

This algorithm checks at each location in $S$ to tell whether the substring at that location matches $\text{key}$
Recurrences for string search

Algorithms with nested loops require two or more recurrences.

*Match* does character by character comparison:

\[
Match(S_1, S_2) =
\begin{cases}
  \text{true} & \text{if } |S_1| = |S_2| = 0 \\
  \text{false} & \text{if } S_1[1] \neq S_2[1] \\
  Match(S_1[2..|S_1|], S_2[2..|S_2|]) & \text{otherwise}
\end{cases}
\]

*Search* tells whether \(S_2\) can be found in \(S_1\):

\[
Search(S_1, S_2) =
\begin{cases}
  \text{true} & \text{if } Match(S_1, S_2) \\
  \text{false} & \text{if } |S_1| < |S_2| \\
  Search(S_1[2..|S_1|], S_2) & \text{otherwise}
\end{cases}
\]

---

**Insertion-sort (A)**

\[
i \leftarrow 1 \\
\text{while } i < |A| \\
A \leftarrow Insert-ascending(A[1..i], A[i+1]) \\
i \leftarrow i + 1 \\
\text{return } A
\]
Recursive insertion sort

**Insertion-sort (A)**

If $|A| \leq 1$

return $A$

Else

return $\text{Insert-ascending}$

$(\text{Insertion-sort}(A[1 \ldots |A| - 1]), A[|A|])$

This algorithm recursively sorts all but the last element the array and then inserts the last element into the sorted result

---

Recurrence for insertion sort

**Insertion-sort (A) =**

$$
\begin{cases}
A & \text{if } |A| \leq 1 \\
\text{Insert-ascending} & \text{otherwise}
\end{cases}
$$

$$
T_{\text{ins-sort}}(n) =
\begin{cases}
O(1) & \text{if } n \leq 1 \\
O(n) + T_{\text{ins-sort}}(n - 1) & \text{otherwise}
\end{cases}
$$
Bubble sort

Repeat
  \[\text{swap} \leftarrow \text{false}\]
  for \(i \leftarrow 1\) to \(\text{size}(A) - 1\)
    if \(A[i] > A[i + 1]\)
      \(\text{swap} \leftarrow \text{false}\)
      \(\text{swap} \leftarrow \text{true}\)
  until \(\text{swap} = \text{false}\)

- The use of nested loops suggests what about running time, for an \(n\)-element array?
- Challenge: Write a recursive version of Bubble

Bubble sort recurrences

\[\text{Max-at-right}(A) = \]
\[
\begin{cases} 
  A & \text{if } |A| \leq 1 \\
\end{cases}
\]

- Returns \(A\), modified so that max element is at right

\[\text{Bubble} (A) = \]
\[
\begin{cases} 
  A & \text{if } |A| \leq 1 \\
  \text{Bubble} (\text{Max-at-right}(A) [1 \ldots |A|-1]) + \text{Max}(A) & \text{otherwise}
\end{cases}
\]

- Returns Bubble-sort of \(A\), minus the maximum, followed by max element
### Selection Sort, recursive version

**Selection-sort (A, start, size)**

If size > 1

\[ i \leftarrow \text{min-index}(A[start..size]) \]

Swap (A[start], A[i])

Return **Selection-sort (A, start + 1, size)**

else return A

---

**Time-complexity recurrence:**

\[
T_{\text{sel-sort}}(n) =
\begin{cases}
1 & \text{if } n = 1 \\
\theta(n) + 5 + T_{\text{sel-sort}}(n - 1) & \text{if } n > 1
\end{cases}
\]

\[
T_{\text{sel-sort}}(n) = \theta(n^2)
\]

---

### Strategy of \(O(n^2)\) sorting algorithms

- Bubble, Selection and Insertion sorts have nested loops that in different ways move items from the unsorted part to the sorted part
- Removing and adding steps are \(O(n)\), hence these sorts are \(O(n^2)\)

**Sort(A)**

For \(i \leftarrow 1\) to \(n - 1\)

remove one element from unsorted part of A

add it to sorted part
Matrix multiplication

• Given two 2-dimensional matrices, their product is a matrix whose cells are each the sum of the products of several cells chosen from both matrices

• Given square matrices A and B, let C be the product;

\[ C[i, j] = A[i, 0] \cdot B[0, j] + \ldots \]
\[ A[i, k] \cdot B[k, j] + \ldots \]
\[ A[i, n-1] \cdot B[n-1, j] \]

for \( i, j \leq n - 1 \)

Matrix multiplication algorithm

\textbf{Matmult} (M[1..hM, 1..vM], P[1..hP, 1..vP])

> Pre: \( vM = hP \)

for \( i \leftarrow 1 \) to \( vM \)

for \( j \leftarrow 1 \) to \( hM \)

\( Y[i, j] \leftarrow 0 \)

for \( k \leftarrow 1 \) to \( hM \)

\( Y[i, j] \leftarrow Y[i, j] + M[j] \ldots \)

return \( Y \)
3. Algorithm analysis; brute force

Brute-force closest-pair

- **Problem:** From a set of pairs of points on a coordinate graph, select the pair of points that are in closest proximity,
- **Brute-force solution:**
  - For each pair of points, compute distance
  - find points that generate the minimum of this set

```
Closest pair (P, i, j)
If n < 2 throw exception
max-dist ← Distance(P, 1, 2)
y ← (1, 2)
for i ← 1 to n – 1
  for j ← i + 1 to n
    if Distance(P, i, j) < max-dist
      max-dist ← Distance(P, i, j)
      y ← (i, j)

Return y
```

```
Distance (A, i, j)
```
Brute-force convex hull

• *Problem:* Find smallest convex polygon that contains all of a set of points
• *Solution:* Generate all segments joining pairs of points; select those segments for which all other points are on same side of that segment

Solving recurrences for nested-loop quadratic-time algorithms

\[ T(n) = \begin{cases} 1 & \text{if } n = 1 \\ \theta(n) + T(n - 1) & \text{otherwise} \end{cases} \]

*Solution:* $\theta(n^2)$, because $n$ steps are needed for each of $(n-1)$ levels of recursion

*Examples:* bubble sort, selection sort, insertion sort
Servant Conjecture (Keil)

- Let \( T(n) = T(n - k) + f(n) \), with \( k \) constant, \( f(n) \in \Theta(n^d) \)
- Then \( T(n) \in \Theta(n f(n)) \)
- Examples:
  - Linear search is \( \Theta(n) \)
  - Bubble, Selection, Insertion sorts are \( \Theta(n^2) \)
- The Servant Conjecture is useful, and is probably easily proven – try it!

4. Exhaustive search

How would you solve these problems?

- SAT
- Closest pair
- Set partition
Subtopic objective

3.4a Describe an algorithm that performs an exhaustive search
3.4b Code and test an exhaustive-search algorithm*†

Exhaustive-search algorithms

• Generate each possible solution to problem; select one that satisfies constraint
• *Optimization version*: Select best-valued one that satisfies constraint
• Example: Traveling Salesperson Problem, finding least-cost path through a set of cities
• Solution: Compare costs of all paths
• Analysis: running time is exponential in number of cities
Exponential-time algorithms

- Exhaustive search often requires time exponential in the size of the problem
- Examples: Towers of Hanoi/Brahma, TSP, SAT, knapsack problem
- Double recursion (e.g., Fibonacci) may take exponential time
- These algorithms may be necessary to solve problems in which combinatorial explosion occurs

Longest common subsequence

- Problem: Given sequences $x_1$, $x_2$, what is the longest string $y$ s.t. $y$ is a subsequence of both $x_1$ and $x_2$? (e.g., “dab” is a subsequence of “database” and “gadabout”)
- Brute-force solution: For each subsequence of $x_1$, search for it in $x_2$, storing the longest found as this process continues and replacing it as a longer common subsequence is found
- Time: exponential, because of the number of subsequences
Knapsack Problem

• *Given:* $n$ items valued $v_1 \ldots n$ and with weights $w_1 \ldots n$
• *Problem:* find maximum-valued subset of weighted items under a given total weight.
• *Brute-force solution:* Find total weight of each subset, find subset with maximum value where weight does not exceed maximum weight
• *Analysis:* time exponential in number of elements of set, since the power set of an $n$-set has $2^n$ elements

Fibonacci is doubly recursive

**Fibonacci** ($n$)

if $n < 2$
  return 1
else
  return $Fibonacci$ ($n - 1$) + $Fibonacci$ ($n - 2$)

• Running time: $\Theta(2^n)$
Solving time recurrences for doubly recursive algorithms

\[ T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
1 + 2T(n-1) & \text{otherwise}
\end{cases} \]

**Solution:** \( O(2^n) \), because at each of \((n-1)\) recursive steps the amount of time required doubles

*Example:* Towers of Hanoi/Brahma

---

**Binomial coefficients**

- \( C(n, k) \) (combinations of \( n \) items taken \( k \) at a time) is also called a *binomial coefficient*, is computed by Pascal’s Triangle, and is defined by Pascal’s formula:

  \[ C(n, k) = \begin{cases} 
1 & \text{if } k = 0 \text{ or } k = n \\
C(n-1, k-1) + C(n-1, k) & \text{otherwise}
\end{cases} \]

- *Binomial theorem:* \((a + b)^n = \sum_{k=0}^{n} C(n, k) a^{n-k} b^k\)
3. Algorithm analysis; brute force

**Hanoi** \((source, dest, intermed, ndisks)\)

If \(ndisks > 0\)

\[
Hanoi(source, intermed, dest, ndisks - 1)
\]

Move top disk from \(source\) to \(dest\)

\[
Hanoi(intermed, dest, source, ndisks - 1)
\]

(Move the top \((n - 1)\) disks to the middle peg, then move bottom disk to the goal peg, then move all \((n - 1)\) disks on the middle peg to the goal.)

**Recurrence, expressing number of steps:**

\[
T_Hanoi(n) = \begin{cases} 
1 & \text{if } n = 0 \\
2T_Hanoi(n - 1) + 1 & \text{if } n > 1 
\end{cases}
\]

**Complexity of Hanoi solution**

\[
T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
2T(n - 1) + 1 & \text{if } n > 1 
\end{cases}
\]

- Recurrence suggests that time doubles on each recursive step, so guess running time is \(Q(2^n - 1)\)

- Proof by induction:
  1. (Base) It works for \(n = 1\): \(2^n - 1 = 2 - 1 = 1\)
  2. If \(T(n - 1) = 2^n - 1 - 1\) then \(T(n) = 2^n - 1\):
     \[
     T(n) = 2T(n - 1) + 1 \quad \text{Per recurrence} \\
     = 2(2^n - 1 - 1) + 1 \quad \text{Guess} \\
     = 2^n - 2 + 1 \quad \text{Add exponents} \\
     = 2^n - 1 \quad \text{(QED)}
     \]
3. Algorithm analysis; brute force

Printing all \( n \)-digit numerals

\[ \text{Numerals}(n) = \begin{cases} 
\lambda & \text{if } n = 0 \\
"0" + \text{Numerals}(n - 1) + \\
"1" + \text{Numerals}(n - 1) & \text{otherwise}
\end{cases} \]

\[
T(n) = \begin{cases} 
1 & \text{if } n = 1 \\
1 + 2T(n - 1) & \text{otherwise}
\end{cases}
\]

\[ = \Theta(2^n) \]

Problems with formulas in logic

Given a propositional logic formula,

- Does it hold, given a set of variable assignments? 
  \( O(n) \) brute-force solution: evaluate the formula under the given interpretation
- Is it a tautology (always valid)?
- Is it a contradiction (never valid)?
- Is it satisfiable?

Brute-force solution: Generate a truth table, with \( 2^n \) rows for \( n \) variables; check for \textit{true}
Satisfiability brute-force solution

**Satisfiable**(\(\phi, t\))

> Pre: Phi is a formula in propositional logic,
> with truth assignments \(t_1, t_2, \ldots t_n\),
> \(y \leftarrow false\)
> for \(i \leftarrow 1\) to \(2^{|t|}\)
> > LI: \(y\) is true iff \(\phi\) with var asgt indicated by
> > some bit pattern in \((0 \ldots i - 1)\) evaluates to true
> if \(eval(\phi, i) = true\)
> > \(y \leftarrow true\)
> return \(y\)
> > Post: \(y\) is true iff \(\phi\) with var asgt indicated by
> > some bit pattern in \((0 \ldots 2^{|t|})\) evaluates to true

Recurrence for SAT

- Let \(E\) be a sentence in propositional logic using
  Boolean variables \(x_1, x_2, \ldots, x_n\)
- \(\text{SAT}(E) =\)
  \[
  \begin{cases}
  true & \text{if } nvars(E) = 0 \land eval(E) = true \\
  false & \text{if } nvars(E) = 0 \land eval(E) = false \\
  \forall i \leq n \text{ SAT}(E[x_i / true]) \lor \\
  \forall i \leq n \text{ SAT}(E[x_i / false]) & \text{otherwise}
  \end{cases}
  \]
- [to be verified!]
- **Strategy**: try each branch of decision tree
**Brute-force set partition**

- *Problem:* To decide if set $S$ of natural numbers can be partitioned into two subsets, $A$ and $B$, such that the sum of elements of $A$ is the same as the sum of elements of $B$

- *Solution:* examine each subset $A$ of $S$, comparing the sum of its elements to the sum of elements of $B$ ($= S - A$)

- *Analysis:* a set $S$ has $2^{|S|}$ subsets, so $\Theta(2^n)$

---

**Brute-force solution to independent-set**

- An *independent set* of vertices has no pair of elements in conflict (i.e., not adjacent to each other)

- As with set partition, a natural solution is to check each subset to see if it satisfies the definition of an independent set

- *Running time:* $\Theta(2^n)$
Optimization problems

- Many problems require finding the best solution in a very large \(O(2^n)\) set of possible solutions
- Brute force (exhaustive search) approach enables simple design with possibility of long running times
- Divide and conquer, greedy, dynamic programming, and approximation approaches are often more efficient

References


