**Topic 4:** Brute-force algorithms

1. Numeric computations
2. Vector traversal
3. Sorting
4. Exhaustive search

*Reading: Sec. 11.1; Levitin Ch 2*

---

**Brute force**

- **Definition:** “a straightforward approach, usually based directly on the problem’s statement…” (Levitin, p. 97)
- **Advantages:**
  - applicable to a wide range of problems
  - simple to design
  - may be useful to solve small problem instances
- **Disadvantage:** inefficient in general case
- **Topic objective:** Explain the brute force design approach to algorithmic problems, and code solutions designed under it
**Objectives**

4a. Write a recurrence that defines the function computed by a brute-force algorithm

4b. Write and solve a time recurrence for a brute-force algorithm

4c. Explain and use the brute force approach, and analyze solutions designed under it

---

**1. Numeric computations**

**Multiplication and exponentiation**

\[
\text{Product} \ (a,b) = \begin{cases} 
a & \text{if } b = 1 \\
a + \text{Product} \ (a, b-1) & \text{otherwise}
\end{cases}
\]

\[
\text{Pow} \ (a,b) = \begin{cases} 
1 & \text{if } b = 0 \\
a \times \text{Pow} \ (a, b-1) & \text{otherwise}
\end{cases}
\]
Decrease by one

Decrease-by-one algorithms have recurrences that reduce the quantity in the recursive case by one

**Factorial** \((n)\)

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

**Fibonacci** \((n)\)

if \(n > 0\)
    return 1
else
    return \(\text{Fibonacci} (n - 1) + \text{Fibonacci} (n - 2)\)

Solving recurrences: decrease-by-one

- 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
- \(T(n) = T(n - 1) + f(n)
    = T(n - 2) + f(n - 1) + f(n) \ldots\)
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:
  \[ C[i, j] = A[i, 0]B[0, j] + \ldots \]
  \[ A[i, k]B[k, j] + \ldots \]
  \[ A[i, n-1]B[n-1, j] \] for $i, j \leq n - 1$

Matrix multiplication algorithm

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 $hM$
      $Y[i, j] \leftarrow Y[i, j] + M[j]$

return $Y$
2. Vector traversal

Linear search

\[ \text{Search}(A, x) = \begin{cases} 
  \text{false} & \text{if } |A| = 0 \\
  \text{true} & \text{if } A[1] = x \\
  \text{Search}(A[2..|A|], x) & \text{otherwise}
\end{cases} \]

Running time:

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

Adding elements of an array

\[ \text{Array-sum}(A) \]

\[
\begin{align*}
  i & \leftarrow 1 \\
  y & \leftarrow 0 \\
  & \quad y \leftarrow y + A[i] \\
  & \quad i \leftarrow i + 1 \\
\end{align*}
\]
Analysis of \textit{array-sum}

\textbf{Recurrence for Array-sum:}

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

\textbf{Time recurrence:}

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

---

\textbf{Brute-force string search}

Finds location of first occurrence of \textit{key} in \textit{S}

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

\begin{algorithmic}
\For{$i \leftarrow 1$ to $n - m + 1$ do}
\State $j \leftarrow 1$
\While{$j \leq m$ \&\& $key[j] = S[i + j]$ do}
\State $j \leftarrow j + 1$
\EndWhile
\If{$j > m$} return $i$
\EndIf
\EndFor
\EndAlgorithmic

Return $-1$
Recurrences for string search

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

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

Longest common subsequence

- **Problem:** Given sequences \(x_1, x_2\), what is the longest subsequence \(y\) s.t. \(y\) is a subsequence of both \(x_1\) and \(x_2\)?

- Elements of subsequences are not necessarily contiguous, e.g., “dab” is a subsequence of “database”

- **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
Complexity of array insertion

\[ \text{Insert-ascending} \ (A, x) \]
\[ i \leftarrow |A| \]
\[ \text{while } i > 1 \text{ and } x < A[i] \]
\[ A[i] \leftarrow A[i - 1] \]
\[ i \leftarrow i - 1 \]
\[ A[i] \leftarrow x \]
\[ \text{Return } A \]

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

Recursive array insertion

\[ \text{Insert-ascending} \ (A, x) \]
\[ \text{If } |A| = 0 \]
\[ \text{return } \langle x \rangle \]
\[ \text{else} \]
\[ \text{if } x < A[1] \]
\[ \text{return } \langle x \rangle + A \]
\[ \text{else} \]
\[ \text{return } \langle A[1] \rangle + \]
\[ \text{Insert-ascending} (A[2..|A|], x) \]
4. Brute force

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

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

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

\[= O(n)\]

3. Sorting

**Insertion sort**

\[
i \leftarrow 1
\]

While \( i < |A| \)

\[
\text{Insert-ascending}(A[1 \ldots i], A[i + 1])
\]

\[
i \leftarrow i + 1
\]

Return A

**Complexity analysis:** \(O(\ ?)\) Why?
Recursive insertion sort

**Insertion-sort (A)**

If $|A| \leq 1$

return $A$

Else

return `Insert-ascending`

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

\[
T_{\text{ins-sort}}(n) = \begin{cases} 
O(1) & \text{if } n \leq 1 \\
O(n) + T_{\text{ins-sort}}(n - 1) & \text{otherwise}
\end{cases}
\]
Selection Sort, recursive version

**Selection-sort** \((A, \text{start}, \text{size})\)

If \(\text{size} > 1\)

1. Find lowest value in \(A[\text{start}..\text{size}]\)
2. Swap it with \(A[\text{start}]\)
3. Return **Selection-sort** \((A, \text{start} + 1, \text{size})\)

else return \(A\)

**Time-complexity recurrence:**

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

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

Bubble-sort\((A)\)

**Precondition:** \(A\) is an array

\(\text{num\_passes} \leftarrow 0\)

Repeat

- **Invariant:** Rightmost (num\_passes) elements are in ascending order
- \(\text{swapped} \leftarrow \text{false}\)

for \(i \leftarrow 1\) to \(\text{size}(A) - 1\)

- **Invariant:** \(A[i]\) is not smaller than any in \(A[0..i - 1]\)
- if \(A[i] > A[i + 1]\)
  - \(\text{swap}(A[i], A[i + 1])\)
  - \(\text{swapped} \leftarrow \text{true}\)

\(\text{num\_passes} \leftarrow \text{num\_passes} + 1\)

until \(\text{swapped}\) is \(\text{false}\)

**Postcondition:** \(A\) is in ascending order
### Bubble sort

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

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

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

- Returns \(\text{Bubble-sort}\) of \(A\), minus the maximum, followed by max element

### Solving recurrences: quadratic

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

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

**Examples:** bubble sort, selection sort, insertion sort
4. Brute force

5. Exhaustive search

• Generate each element of a problem domain, select one that satisfies constraint
• Optimization version: Select best-valued one that satisfies constraint
• Example: Traveling Salesperson Problem, finding minimum-cost path through all cities on a map.
• 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
4. Brute force

Brute-force closest-pair

- **Problem:** From a set of all pairs of points on a coordinate graph, select the pair of points that are in closest proximity,
- **Solution:** For all pairs of points, compute distances, find points that generate the minimum of this set

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

Distance (A[i], j)
Return sqrt(
    (A[i].x - A[j].x)^2 +
    (A[i].y - A[j].y)^2
)
4. Brute force

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

Knapsack Problem

- **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
Hanoi \((\text{source, dest, intermed, ndisks})\)

If \(\text{ndisks} > 0\)

\[
\begin{align*}
\text{Hanoi} & (\text{source, intermed, dest, ndisks}-1) \\
\text{Move top disk from source to dest} \\
\text{Hanoi} & (\text{intermed, dest, source, ndisks}-1)
\end{align*}
\]

(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:

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

Complexity of Hanoi solution

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

- Recurrence suggests that time doubles on each recursive step, so guess running time is \(\Theta(2n - 1)\)
- Proof by induction:
  1. It works for \(n = 1\): \(2^n - 1 = 2 - 1 = 1\)
  2. If \(T(n - 1) = 2^n - 1 - 1\) then \(T(n) = 2^n - 1\):
     \[
     T(n) = 2T(n - 1) + 1 \\
     = 2(2^n - 1 - 1) + 1 \\
     = 2^n - 2 + 1 \\
     = 2^n - 1 \quad \text{(QED)}
     \]
Solving double recursion

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

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

*Example:* Towers of Hanoi/Brahma

---

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 (valid under all variable assignments)?
- Is it a contradiction (valid under no assignments)?
- Is it satisfiable?
  
  *Brute-force solution:* Generate a truth table, with \(2^n\) rows for \(n\) variables, see if any rows evaluate to \(true\)
Optimization problems

- Many problems require finding the best solution in a very large (O(2^n)) set of possible solutions
- Brute force (exhaustive search) offers an easily designed but long-running-time approach
- Divide and conquer, greedy, dynamic programming, and approximation approaches are often more efficient

Concepts

- brute force
- brute-force closest-pair
- brute-force convex hull
- brute-force knapsack
- brute-force Traveling Salesperson
- Bubble sort
- Drag-max
- exhaustive search
- Insert-ascending
- insertion sort
- linear search
- matrix multiplication
- Towers of Hanoi (Brahma)
4. Brute force

References


D. Keil classroom work.