Unit 11. Randomized Algorithms

Algorithms

EE3980

Jun. 18, 2020

Quick Sort Revisited

- Algorithm Quick Sort (Algorithm 3.2.5) is shown to have average complexity of $\mathcal{O}(n \lg n)$ and is repeated below.

Algorithm 11.1.1. Quick Sort

```
// Sort A[low : high] into nondecreasing order.
// Input: A[low : high], int low, high
// Output: A[low : high] sorted.
Algorithm QuickSort(A, low, high)
{
    if (low < high) then {
        mid := partition(A, low, high + 1);
        QuickSort(A, low, mid − 1);
        QuickSort(A, mid + 1, high);
    }
}
```

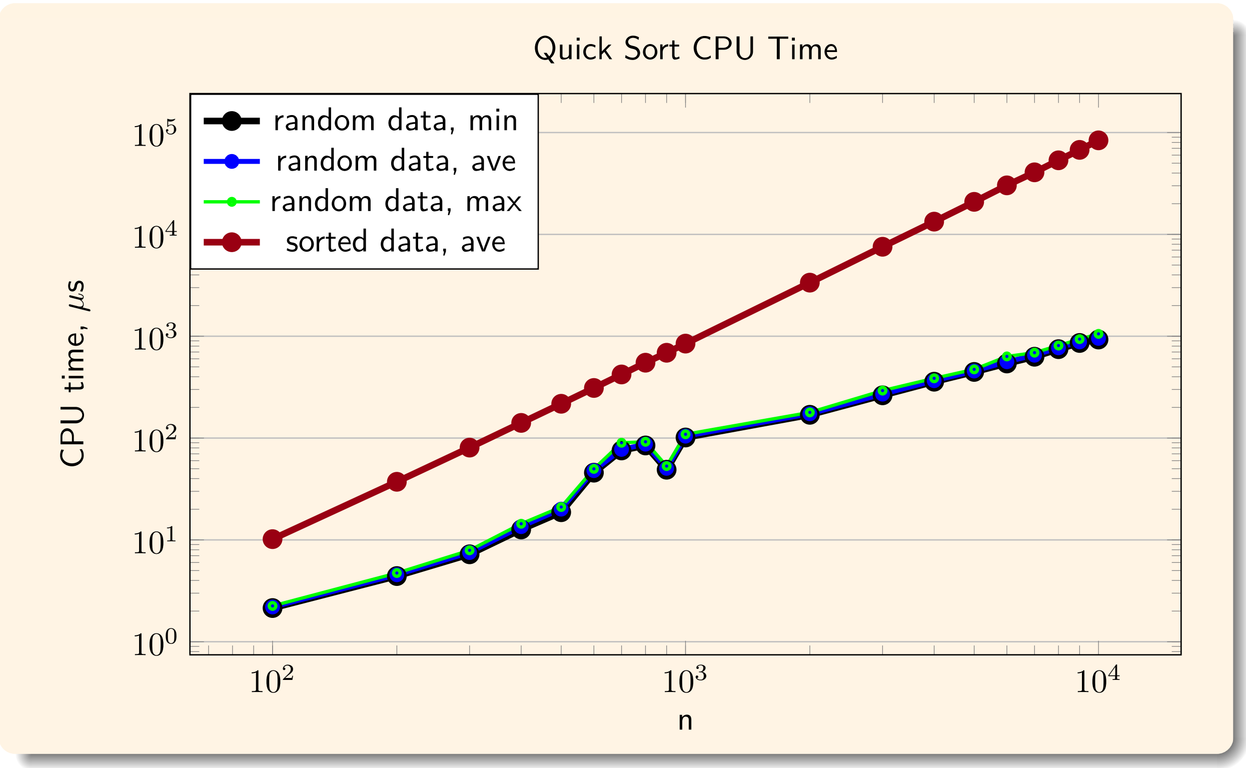
- It is a divide-and-conquer algorithm.
- The divide function Partition is repeated as well.
  - It is also known that Partition has the worst-case complexity of $\mathcal{O}(n^2)$ and average complexity of $\mathcal{O}(n)$.
  - The latter contributes to Quick Sort's $\mathcal{O}(n \lg n)$ complexity.

Partition Algorithm

Algorithm 11.1.2. Partition

```
// Partition A into A[low : mid − 1] ≤ A[mid] and A[mid + 1 : high] ≥ A[mid].
// Input: A, int low, high
// Output: j that A[low : j − 1] ≤ A[j] ≤ A[j + 1 : high].
Algorithm Partition(A, low, high)
{
    v := A[low]; // Initialize
    i := low;
    j := high;
    repeat { // Check for all elements.
        repeat i := i + 1; // Find i such that A[i] ≥ v.
        until (A[i] ≥ v);
        repeat j := j − 1; // Find j such that A[j] ≤ v.
        until (A[j] ≤ v);
        if (i < j) then Swap(A, i, j); // Exchange A[i] and A[j].
    } until (i ≥ j);
    A[low] := A[j]; // Move v to the right position.
    A[j] = v;
    return j;
}
Algorithm Swap(A, i, j)
{
    t := A[i]; A[i] := A[j]; A[j] := t;
}
```


Quick Sort CPU times



- `QuickSort` works well with random data
- However, with pre-sorted data its complexity is shown to be $\mathcal{O}(n^2)$

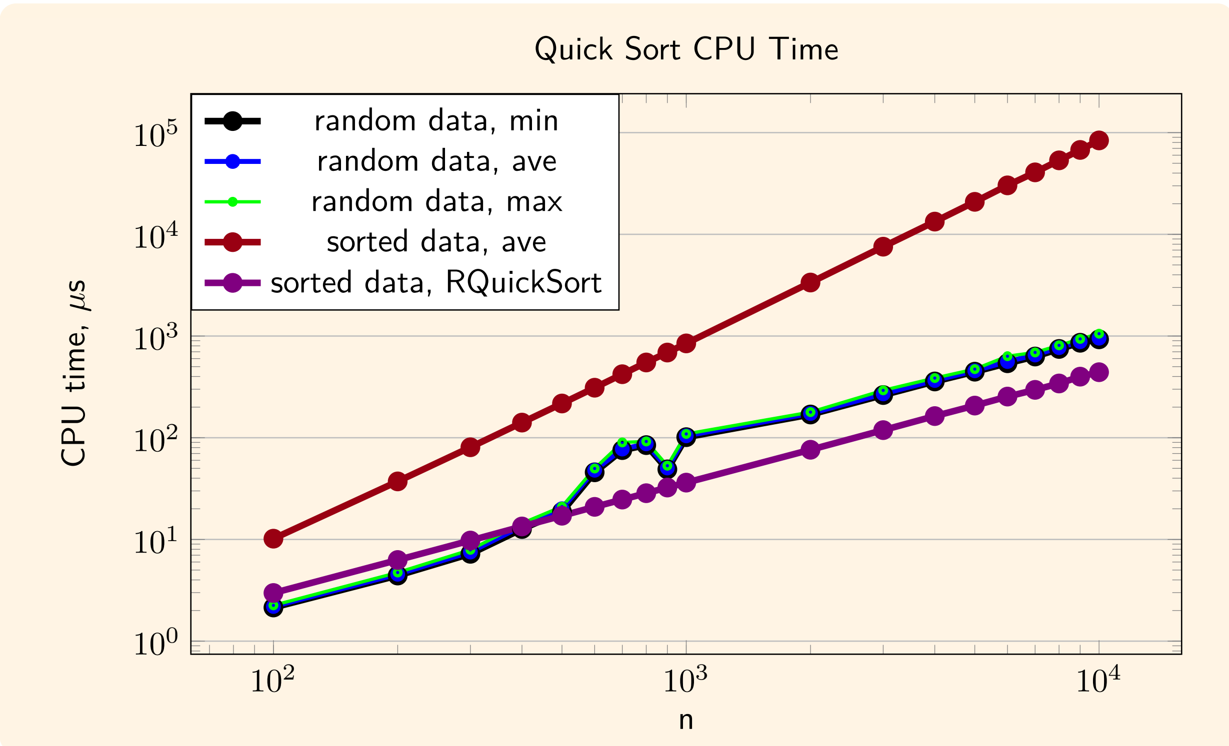
Randomized Quick Sort

- Randomized quick sort (Algorithm 3.2.8), has been proposed to improve the worst-case complexity.

Algorithm 11.1.3. Randomized Quick Sort

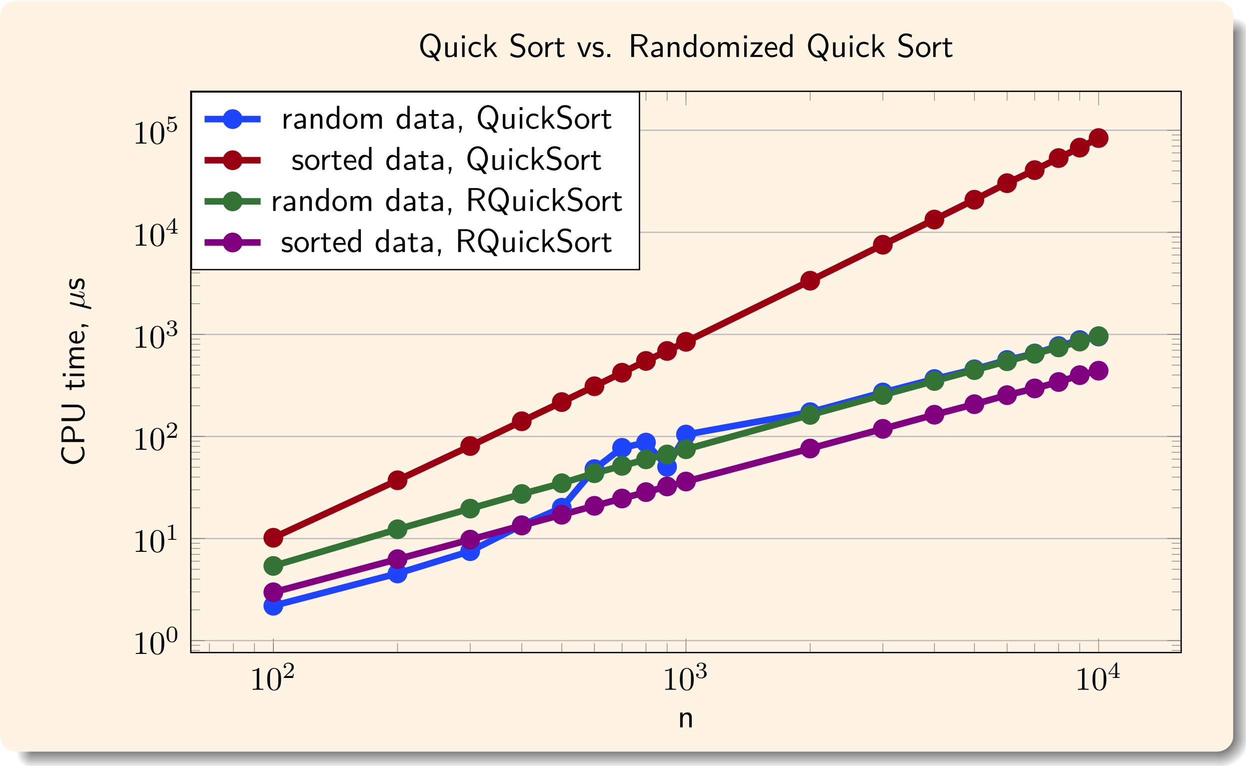
```
// Sort A[low : high] into nondecreasing order.
// Input: A[low : high], int low, high
// Output: A[low : high] sorted.
Algorithm RQuickSort(A, low, high)
{
    if (low < high) then {
        if ((high − low) > 5) then
            Swap(A, low + ( Random() mod (high − low + 1)), low) ;
        mid := Partition(A, low, high + 1) ;
        QuickSort(A, low, mid − 1) ;
        QuickSort(A, mid + 1, high) ;
    }
}
```


Randomized Quick Sort CPU times



- RQuickSort is shown to be very effective in improving the time complexity to $\mathcal{O}(n \lg n)$.

Randomized Quick Sort CPU times



- For random data, RQuickSort and QuickSort have similar CPU times.
- Randomized Quick Sort maintains worst-case complexity to $\mathcal{O}(n \lg n)$.
- Randomized algorithms can be effective in some applications.

Randomized Selection Algorithm

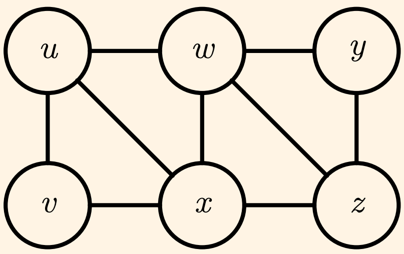
- The Partition algorithm is also used in Select1, Algorithm (3.3.1).
- Similar randomization technique, line 5 of Algorithm RQuickSort can be applied to improve performance.
- Overall average complexity does not change.
- But, CPU tends to get better with randomization.
  - Chance of getting worst-case performance is very small.
  - Smaller for larger n.

Min-Cut Problem

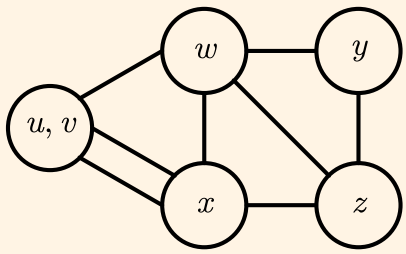
- Given a undirected graph, $G(V, E)$, $|V| = n$, and $|E| = e$, an edge cut, or cut, in G is a subset $C \subset E$ such that C's removal disconnects G into two or more components.
- A minimum cut is a cut with minimum $|C|$.
- Given an edge $(u, v) \in E$, $u, v, \in V$, (u, v) is contracted if vertices u and v are merged into one, all edges connecting u and v are deleted, and all other edges are retained.
- Note that contraction of an edge may result in multiple edges connecting two vertices, but no self-loops, so G may become a multigraph after contraction.

Edge Contraction and a Cut

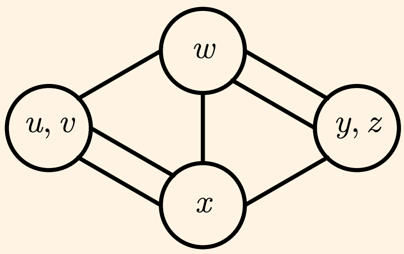
- Using edge contraction to find a cut set.



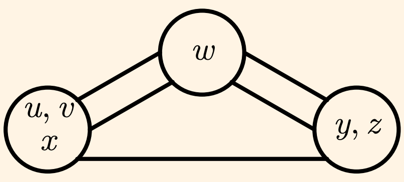
Graph $G(V, E)$.



(u, v) contracted.

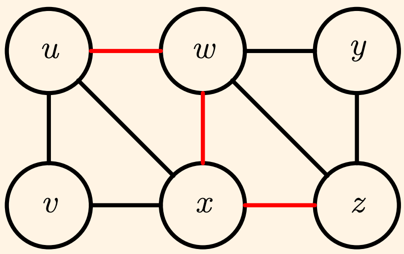


(y, z) contracted.



(u, x) contracted.

(w, y) contracted.



A cut set found.

- Though a cut set is found, it is not the minimum cut set.

Min-Cut Algorithm

- Using edge contraction, we can find a cut set.
- The following randomized algorithm tries to find a minimum cut set.

Algorithm 11.1.4. Min Cut

```
   // Find min-cut given a graph.
   // Input: G(V, E)
   // Output: min-cut set C ⊂ E.
1  Algorithm MinCut(V, E, C)
2  {
3      C = E; // Initialize cut set to E.
4      for i := 1 to r do { // repeat r times.
5          V' := V; // Initialize V' and E'.
6          E' := E;
7          while (|V'| > 2) do { // Contract until two vertices remaining.
8              choose (u, v) ∈ E' randomly ;
9              Contract(V', E', (u, v)) ; // Perform contraction.
10         }
11         if (|E'| < |C|) then C := E' ; // E' is a cut set.
12     }
13 }
```


Min-Cut Algorithm Analysis

- When only two vertices remaining in V' (line 7 of (Algorithm 11.1.4), E' is a cut set.
- We will analyze the probability of finding the minimum cut of the inner loop, lines 7-10, of the MinCut algorithm.
- Assuming $|C| = k$, that is, there are k edges in C, then
  1. The minimum vertex degree is k, otherwise removing a smaller number of edges would isolate the vertex which contradicts to the assumption.
  2. The minimum number of edges is then $kn/2$ for G is a undirected graph.
- Since C is the min-cut, the first edge selected cannot be in C, and the probability of not selecting min-cut edge is

$$1 - \frac{k}{kn/2} = 1 - \frac{2}{n}. \tag{11.1.1}$$

- By the same reason, the probability of not selecting the a min-cut edge on the 2nd selection is

$$1 - \frac{k}{k(n-1)/2} = 1 - \frac{2}{n-1}. \tag{11.1.2}$$

Min-Cut Algorithm Analysis, II

- And, for the i-th selection the probability of not selecting a min-cut edge is

$$1 - \frac{k}{k(n-i+1)/2} = 1 - \frac{2}{n-i+1}. \tag{11.1.3}$$

- The loop terminates when there are two vertices left, with $i = n-2$, and the probability of not selecting edges in C is

$$1 - \frac{k}{k(n-(n-2)+1)/2} = 1 - \frac{2}{3}. \tag{11.1.4}$$

- All conditions, Eq. (11.1.1 – 11.1.4), must be met and we have the probability of getting the min-cut set as

$$\begin{aligned} P(C = \text{min-cut}) &= (1-\frac{2}{n})(1-\frac{2}{n-1})\cdots(1-\frac{2}{3}) \\ &= \frac{n-2}{n}\cdot\frac{n-3}{n-1}\cdot\frac{n-4}{n-2}\cdots\frac{2}{4}\cdot\frac{1}{3} \\ &= \frac{2}{n(n-1)} \\ &< \frac{2}{n^2}. \end{aligned} \tag{11.1.5}$$

Min-Cut Algorithm Analysis, III

- Thus, the probability of not getting the min-cut in one iteration is

$$1 - \frac{2}{n(n-1)} \geq 1 - \frac{2}{n^2}. \tag{11.1.6}$$

The equality holds for $n \gg 1$.
- The inner loop is repeated r times, and the probability of not getting the min-cut is then

$$(1 - \frac{2}{n(n-1)})^r \geq (1 - \frac{2}{n^2})^r. \tag{11.1.7}$$

Conversely, the probability of gettting min-cut is

$$1 - (1 - \frac{2}{n(n-1)})^r \leq 1 - (1 - \frac{2}{n^2})^r. \tag{11.1.8}$$

- Setting $r = \frac{n^2}{2}$, assuming large n we have the probability of getting the min-cut be

$$1 - (1 - \frac{2}{n^2})^{n^2/2} = 1 - \frac{1}{e}. \tag{11.1.9}$$

The last equation comes from Eq. (1.4.21).
- The overall time complexity is
  - Each `Contract` takes $\mathcal{O}(n)$ operations.
  - $n-2$ `Contact` performed for one iteration, $\mathcal{O}(n^2)$.
  - Repeating $n^2/2$ times result in $\mathcal{O}(n^4)$ complexity.

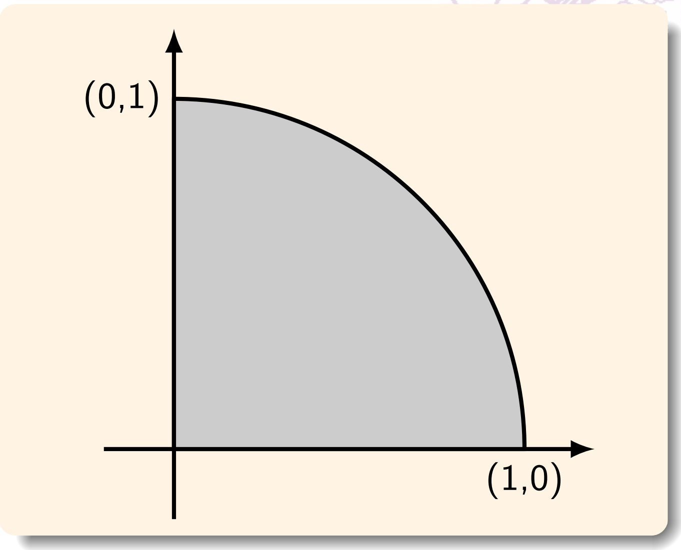
Las Vegas vs. Monte Carlo algorithms

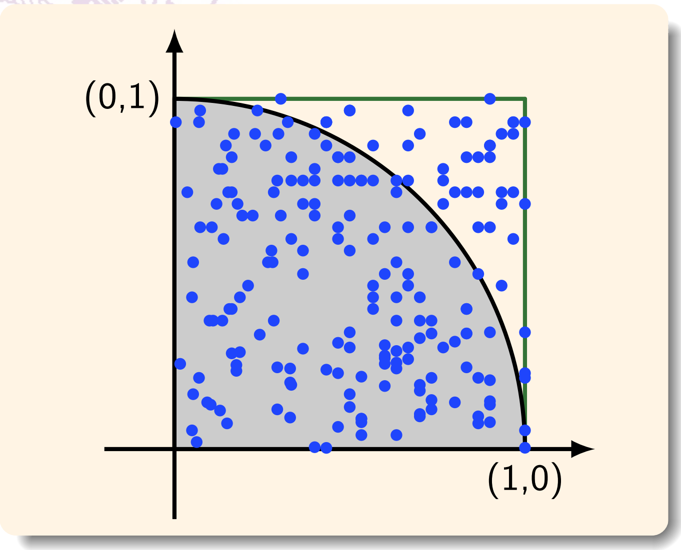
- In this MinCut randomized algorithm, Algorithm (11.1.4), to find a minimum cutting set, Eq. (11.1.8) shows as the number of iterations increases, the probability of getting the right answer increases as well.
- At the end of each iteration, we have a cut set. But, it is not necessarily the minimum cut set.
- The algorithm can stop for any integer number of iterations, but not guaranteeing the optimal answer.
- This is called the Monte Carlo type of randomized algorithm.

- In contrast, the Randomized Quick Sort, Algorithm (11.1.1) always produces the right answer.
- The latter is called the Las Vegas type of randomized algorithm.

Monte Carlo Integration

- Closed form solution for integration can be difficult to carry out.
- Numerical integration is usually preferred.
- An alternative approach is the Monte Carlo method.





Monte Carlo Integration – Algorithm

- Given a function $f(x)$, the definite integral is to be solved for.

$$\int_{x=a}^{b} f(x)\,\mathrm{d}x \tag{11.1.10}$$

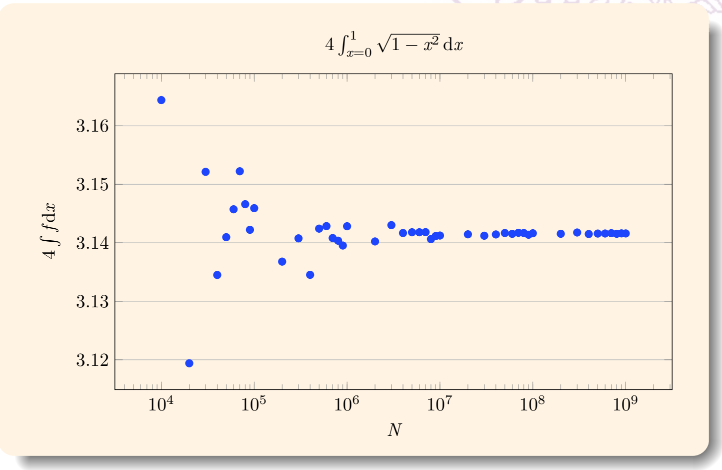
It is assumed that $0 \le f(x) \le h$, $a \le x \le b$.

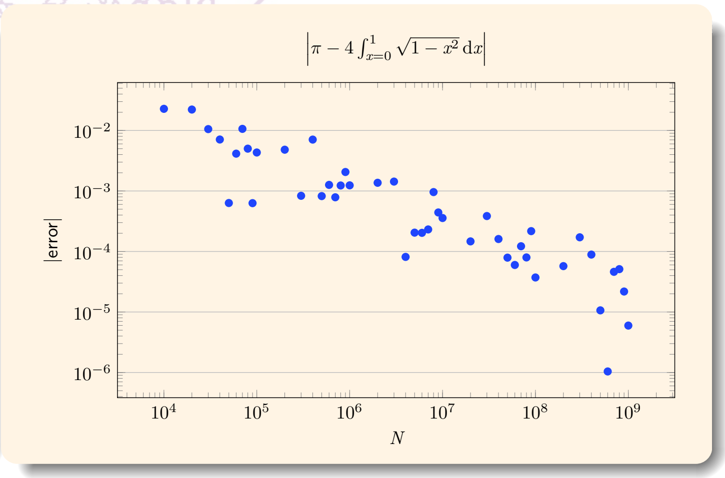
Algorithm 11.1.5. Monte Carlo Integration

```
   // To find ∫_{x=a}^{b} f(x) d, 0 ≤ f(x) ≤ h.
   // Input: a, b, h
   // Output: integral.
 1 Algorithm Integrate(a, b, h)
 2 {
 3     I := 0;
 4     for i := 1 to N do {
 5         x := rand(a, b);
 6         y := rand(0, h);
 7         if y ≤ f(x) then I := I + 1;
 8     }
 9     return (b − a)×h×I/N;
10 }
```


Monte Carlo Integration – Analysis

- It is also assumed rand(a, b) function generates a random number uniformly in the range $[a, b]$.
- The loop, lines 4-8, executes N times, thus the time complexity is $\Theta(N)$.
- As N increases, the function should return value approaches the real integral.

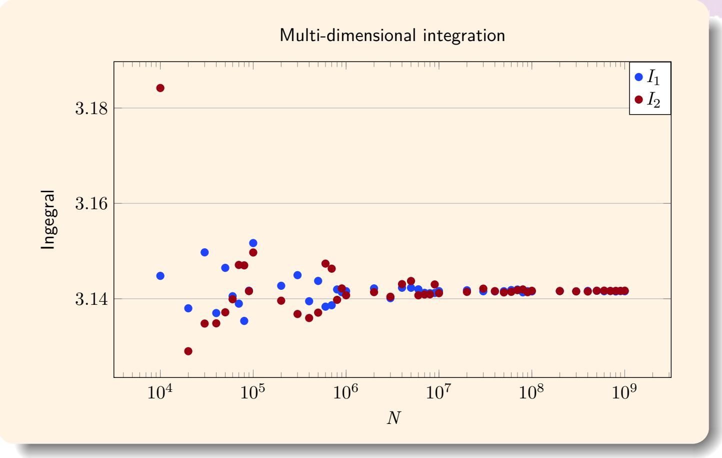


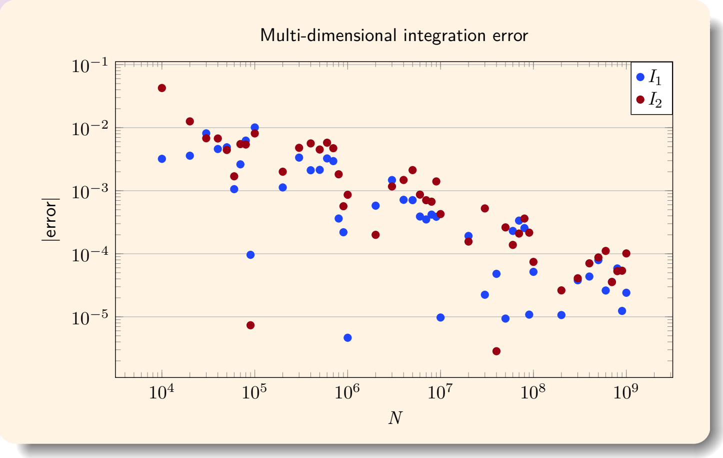


Multi-dimensional Integration

- Multi-dimensional integration can also be implemented easily using Monte Carlo approach.
- For example,

$$I_1 = 4\int_{x=0}^{1} \sqrt{1-x^2}\,\mathrm{d}x, \qquad I_2 = 6\int_{x=0}^{1}\int_{y=0}^{1} \sqrt{1-x^2-y^2}\,\mathrm{d}x\,\mathrm{d}y$$





Monte Carlo Integration and Random Function

- Monte carlo integration algorithms are very easy to implement.
- Solution accuracy appears to increase with number of samples (N).
  - Error decreases linearly with N, but not monotonically.
- Uniformity of random number distribution affects the accuracy.
  - Choosing a good random number generator is very important.
- Multi-dimensional integration is easily generalized from 1-dimensional integration.
- Monte carlo integration is more effective in multi-dimensional integration problems.
- Lower dimension integrations can use more effective deterministic formulas, such as Newton-Cotes formulas.

Matrix Verification Problem

- Given three $N \times N$ matrices, A, B and C, where C approximates $A \times B$, and we need to find if $C = A \times B$.
- Brute force approach
  1. Find $D = A \times B$,
  2. Check if $C[i,j] = D[i,j]$, $1 \le i,j \le N$.
- Step 1 is $\Theta(N^3)$ since $D[i,j] = \sum_{k=1}^{N} A[i,k] \times B[k,j]$

  and there are N^2 elements in D.
- Step 2 is $\Theta(N^2)$ time due to N^2 elements.
- Thus, brute force approach is $\Theta(N^3)$.
  - For large N it can be very time consuming.

Matrix Verification – Monte Carlo Approach

- Matrix verification problem can be solved using Freivald's technique.

Algorithm 11.1.6. Matrix Verification

```
// Given N × N matrices A, B and C, check if C = A × B.
// Input: A, B, C, N
// Output: 1: if C = A × B, 0: otherwise.
Algorithm MatVerify(A, B, C, N)
{
    for i := 1 to N do // Generate random vector r, r[i] = 0 or 1.
        if RAND(0, 1) < 0.5 then r[i] = 0;
        else r[i] = 1;
    x := A×(B×r); // Two matrix-vector multiplications.
    y := C×r; // Matrix vector multiplication.
    for i := 1 to N do // Check if x = y.
        if x[i] ≠ y[i] then return 0;
    return 1;
}
```

- r is an N-vector with $r[i] = 0$ or 1, $1 \le i \le N$.
- $\mathtt{RAND}(0,1)$ generates a random number uniformly in the range $[0,1]$.

Matrix Verification – Analysis

- In the preceding algorithm, loops on lines 3-5 and 8-9 both execute $\mathcal{O}(N)$ times.
- Lines 6 and 7 consist of 3 matrix-vector multiplications, $\Theta(N^2)$.
- Thus, overall complexity is $\Theta(N^2)$.
  - This is much faster than the brute force approach.

Theorem 11.1.7.

Given three $N \times N$ matrices A, B and C, $A \times B \neq C$ and a randomly generated N-vector r, $r[i] = 0$ or 1, $1 \leq i \leq N$, then the probability that Algorithm `MatVerify` returns 1 is less than or equal to $1/2$.

Proof. Assume that C and $A \times B$ differs only at $C[i,j]$, then $r[j]$ needs to be 1 such that `MatVerify` would return 0. The chance that $r[j] = 1$ is $1/2$, thus proves the theorem. □

- One call to Algorithm `MatVerify` has the failure rate of $1/2$.
- Repeat the algorithm k times one gets the failure rate $(1/2)^k$.
  - The complexity is still $\Theta(N^2)$ for fixed k.
- Thus, this approach can very effective in verifying matrix equality problem.

Summary

- Quick sort revisited
  - Average-case complexity vs. worst-case
- Randomized quick sort
  - Avoiding worst-case complexity
  - Las Vegas type of randomized algorithm
- Graph min-cut problem
- Randomized integration algorithms
- Matrix verification problem
- Monte Carlo type of randomized algorithms
  - Have been used in solving physics problems