Sortering – IN2010: Algoritmer og Datastrukturer

Oversikt og motivasjon • Sortering handler om å ordne en samling elementer etter en bestemt rekkefølge, typisk stigende eller synkende. • Sortering er grunnleggende i informatikk, og brukes i alt fra søk og dataanalyse til grafalgoritmer og prioriteringssystemer. • Effektiv sortering kan drastisk redusere kjøretiden for andre algoritmer, spesielt når man må gjøre mange oppslag eller sammenligninger.

Mål denne uken

1. Definere sorteringsproblemet: ○ Input: En liste eller array med elementer. ○ Output: De samme elementene, men i ønsket rekkefølge.
2. Motivasjon for ulike algoritmer: ○ Enkel implementasjon: Lære logikk og grunnleggende operasjoner. ○ Effektivitet: Forstå tidskompleksitet, O-notasjon og praktisk ytelse. ○ Spesielle situasjoner: Noen algoritmer fungerer bedre når data allerede er delvis sortert, eller når elementene har spesifikke egenskaper.

Sorteringsalgoritmer som skal gjennomgås

1. Bubble Sort ○ Sammenligner og bytter naboelementer. ○ Enkel, men ineffektiv for store datasett ($O(n^2)$).
2. Selection Sort ○ Velger minste element og plasserer det først. ○ Enkelt å forstå, men fortsatt $O(n^2)$.
3. Insertion Sort ○ Setter hvert nytt element på riktig plass i en allerede sortert del av listen. ○ Effektiv for små eller delvis sorterte lister ($O(n^2)$ i verste, $O(n)$ i beste tilfelle).
4. Heapsort ○ Bruker binære heaps til å effektivt finne og fjerne minste/største element. ○ $O(n \log n)$ i alle tilfeller, in-place.
5. Merge Sort ○ Deler listen i to, sorterer hver halvdel, og slår sammen. ○ Stabil, $O(n \log n)$, men krever ekstra plass for merge.
6. Quicksort ○ Velger pivot, deler listen i mindre og større elementer, sorterer rekursivt. ○ Gjennomsnitt $O(n \log n)$, men $O(n^2)$ i verste fall.
7. Bucket Sort ○ Fordeler elementene i "bøtter" etter intervaller og sorterer hver bøtte. ○ Kan være lineær, men krever jevnt fordelte data.
8. Radix Sort ○ Sorterer tall siffer for siffer, vanligvis fra minst til størst signifikant siffer. ○ Kan være lineær for heltall med begrenset antall sifre.

Hvorfor lære alle disse? • Forståelse av kompleksitet: Å se forskjellene i $O(n^2)$ vs $O(n \log n)$ vs $O(n)$ hjelper å velge riktig algoritme for ulike situasjoner. • Praktisk programmering: Mange av disse brukes som byggeklosser i mer avanserte datastrukturer og systemer. • Eksamen og intervjuer: Klassiske sorteringsalgoritmer testes ofte i både teoretiske og praktiske oppgaver.

Sortering – Definisjon av problemet • Mål: Å ordne elementer i en datastruktur etter en bestemt rekkefølge, typisk definert av en total ordning $\leq$. • Formelt: For elementene a og b i datastrukturen skal a komme før b i output dersom $a \leq b$. • Bevaring: Alle elementer fra input skal være til stede i output, ingen elementer går tapt eller dupliseres. • Fokus: Vi konsentrerer oss primært om sortering av arrayer, men prinsippene gjelder også for lister og andre sekvensielle datastrukturer.

Problemer som løses ved sortering

1. Samle ting som hører sammen (Partisjonering) ○ Sortering grupperer elementer med like egenskaper sammen. ○ Eksempel: Ordne personer etter alder, produkter etter kategori eller tall etter størrelse. ○ Dette gjør at man enkelt kan hente ut, analysere eller operere på hele grupper samtidig. ○ Dette kalles ofte partisjonering i informatikk – et sentralt konsept i flere algoritmer, inkludert Quicksort.
2. Matche elementer på tvers av datastrukturer ○ Når vi har flere sekvensielle strukturer (for eksempel to lister), gjør sortering det mulig å finne matchende elementer effektivt. ○ Istedenfor å sammenligne alle elementer parvis (O(n·m)), kan vi kjøre en enkel lineær gjennomgang på sorterte lister (O(n+m)). ○ Eksempel: Finn alle personer som finnes i to forskjellige medlemslister.
3. Effektiv søking ○ Sorterte datastrukturer muliggjør mye raskere søk enn usorterte. ○ Binærsøk på en sortert array gir O(log n) oppslagstid, mens lineært søk i en usortert array er O(n). ○ Sortering legger dermed grunnlaget for effektive algoritmer som krever rask tilgang til data.

Sortering er derfor ikke bare et spørsmål om å ordne elementer pent – det forenkler og effektiviserer andre algoritmersom partisjonering, matching og søk.

Stabilitet i sorteringsalgoritmer • Nøkler: Ofte sorterer vi objekter etter en bestemt nøkkel. ○ Eksempel: Vi har en liste med person-objekter og ønsker å sortere etter navn. • Definisjon av stabilitet: ○ En sorteringsalgoritme er stabil dersom den bevarer rekkefølgen til elementer med lik nøkkel. ○ Formelt: For alle elementer x og y med samme nøkkel k, hvis x kommer før y før sortering, kommer x fortsatt før y etter sortering. • Hvorfor det er viktig: ○ Stabilitet er nyttig når vi sorterer på flere nivåer. § Eksempel: Sorter først personer etter alder, deretter etter navn. Hvis navnesorteringen er stabil, vil alderrekkefølgen bevares blant personer med samme navn. • Implementasjonsavhengig: ○ Om en algoritme er stabil kan avhenge av hvordan den implementeres, selv om prinsippet tillater stabilitet. ○ Eksempel: Insertion sort og Bubble sort er stabile hvis implementert riktig, mens Selection sort er ofte ikke stabil.

In-place sortering • Definisjon: ○ En algoritme er in-place dersom den ikke bruker ekstra datastrukturer for å lagre elementene midlertidig. ○ Det betyr at all manipulering skjer direkte i den eksisterende datastrukturen (for eksempel arrayet som skal sorteres). • Eksempel på in-place algoritmer: ○ Bubble sort – bytter elementer direkte i arrayet. ○ Selection sort – bytter elementer uten ekstra minne. ○ Insertion sort – flytter elementer internt i arrayet. • Ikke in-place: ○ Merge sort – krever ofte et ekstra array for å mellomlagre resultatene, og regnes derfor ikke som in-place i standardimplementasjoner. • Hvorfor det er relevant: ○ In-place algoritmer er minneeffektive, spesielt viktig når datasett er store. ○ Ikke in-place algoritmer kan kreve dobbelt så mye minne, men kan til gjengjeld ha bedre kjøretid eller stabilitet.

Kort sagt: • Stabilitet handler om rekkefølgen til like elementer. • In-place handler om minnebruk og datastrukturmanipulering.

Stabilitet i sorteringsalgoritmer • Sortering på nøkler: ○ Ofte inneholder elementene flere attributter, men vi sorterer på én nøkkel. ○ Eksempel: Person-objekter sortert etter navn, selv om objektene også har alder, adresse osv. • Definisjon av stabilitet: ○ En sorteringsalgoritme er stabil dersom den bevarer rekkefølgen til elementer med lik nøkkel. ○ Formelt: For alle elementer x og y med samme nøkkel k: § Hvis x kommer før y før sortering, kommer x fortsatt før y etter sortering. • Praktisk betydning: ○ Stabilitet er viktig når man ønsker å gjøre flere sorteringer på samme datasett. § Eksempel: Først sortere personer etter alder, deretter etter navn. Hvis navnesorteringen er stabil, vil alderrekkefølgen bevares blant personer med samme navn. • Implementasjonsavhengig: ○ Om en algoritme er stabil kan avhenge

av hvordan den implementeres, selv om algoritmen i teorien kan være stabil. ○ Eksempler: § Bubble sort og Insertion sort er stabile når implementert riktig. § Selection sort er vanligvis ikke stabil.

In-place sortering • Definisjon: ○ En algoritme er in-place dersom den ikke bruker ekstra datastrukturer for å lagre elementene midlertidig. ○ Alle operasjoner skjer direkte i den eksisterende datastrukturen (f.eks. arrayet som skal sorteres). • Eksempler på in-place algoritmer: ○ Bubble sort – bytter elementer direkte i arrayet. ○ Selection sort – bytter elementer uten ekstra minne. ○ Insertion sort – flytter elementer internt i arrayet. • Ikke in-place algoritmer: ○ Merge sort – krever ofte et ekstra array for mellomlagring, og regnes derfor ikke som in-place i standardimplementasjoner. • Hvorfor det er relevant: ○ In-place algoritmer er mer minneeffektive, spesielt for store datasett. ○ Ikke in-place algoritmer kan kreve mer minne, men kan tilby bedre kjøretid eller stabilitet.

Kort sagt: • Stabilitet = bevarer rekkefølgen til like elementer. • In-place = bruker minimalt ekstra minne.

Bubble Sort – Idé • Grunnidé: ○ Bubble sort er en enkel sorteringsalgoritme som fungerer ved å gjentatte ganger gå gjennom arrayet og "boble" de største elementene til slutten. ○ Prosessen gjentas til hele arrayet er sortert, dvs. ingen flere elementer må byttes. • Trinnvis beskrivelse: 1. Gå gjennom arrayet parvis, fra starten til slutten. 2. Hvis et par elementer ikke er i riktig rekkefølge (A[j] > A[j+1]), bytt om plassene. 3. Fortsett å gå gjennom arrayet til en hel gjennomgang ikke krever noen bytter. • Intuisjon: ○ Største element "bobler" opp til høyre (slutten) i hver iterasjon. ○ Den neste største følger etter i neste gjennomgang. ○ Derfor navnet bubble sort.

Bubble Sort – Implementasjon ALGORITHM: BUBBLE SORT Input: Array A med n elementer Output: Sortert array A Procedure BubbleSort(A): for i ← 0 to n-2 do for j ← 0 to n-i-2 do if A[j] > A[j + 1] then A[j], A[j + 1] ← A[j + 1], A[j] • Kommentarer: ○ Den indre loopen sammenligner og bytter elementer. ○ Den ytre loopen sørger for at prosessen gjentas nok ganger til at hele arrayet blir sortert. ○ Optimalisering: Man kan avslutte tidlig hvis ingen bytter skjedde i en gjennomgang, men dette påvirker ikke verste tilfelle.

Bubble Sort – Kjøretidsanalyse • Iterasjoner: ○ Ytre loop: i = 0..n-2 → totalt n-1 iterasjoner. ○ Indre loop: j = 0..n-i-2 → går over færre elementer for hver iterasjon av ytre loop. ○ Totalt antall sammenligninger: $(n-1)+(n-2)+...+1=n(n-1)2$ • Kompleksitet: ○ Verste tilfelle: $O(n^2)$ ○ Selv med tidlig stopp-optimalisering, forblir $O(n^2)$ i verste tilfelle. ○ Bubble sort er derfor ikke effektiv for store datasett. • Egenskaper: ○ Enkel å implementere. ○ Stabil: Bevarer rekkefølgen til like elementer. ○ In-place: Bruker ikke ekstra datastruktur, bytter elementer direkte i arrayet.

Selection Sort – Idé • Grunnidé: ○ I stedet for å "boble" elementer som i Bubble Sort, går Selection Sort gjennom arrayet for å finn det minste elementet i den usorterte delen og plasserer det på riktig sted. ○ Dette gjentas for hver posisjon i arrayet til hele arrayet er sortert. • Trinnvis beskrivelse: 1. Sett i = 0, som markerer starten på den usorterte delen av arrayet. 2. Finn posisjonen minIndex til det minste elementet i arrayet fra i til slutten. 3. Bytt elementet på plass i med elementet på minIndex hvis de ikke er like. 4. Øk i med 1 og gjenta prosessen til i når slutten av arrayet. • Intuisjon: ○ Etter første iterasjon vil det minste elementet ligge på plass 0. ○ Etter andre iterasjon vil det nest minste elementet ligge på plass 1. ○ Prosessen fortsetter, og arrayet vokser gradvis til å bli sortert fra venstre mot høyre. • Visualisering: ○ Usortert del → [i ... n-1] ○ Sortert del → [0 ... i-1] ○ Hver iterasjon flytter ett element fra usortert del til sortert del.

Insertion Sort – Idé • Grunnidé: ○ Insertion Sort bygger opp en sortert del av arrayet ett element om gangen. ○ Det er veldig likt hvordan man sorterer kort i hånden: du tar ett kort og plasserer det på riktig

sted i den sorterte bunken. • Trinnvis beskrivelse: 1. Sett i = 1 (det første elementet i=0 anses allerede som sortert). 2. Ta elementet på plass i og sammenlign med elementene til venstre i den sorterte delen. 3. Flytt alle større elementer til høyre for å lage plass til elementet på riktig posisjon. 4. Sett elementet inn der det hører hjemme. 5. Øk i med 1 og gjenta prosessen til i når slutten av arrayet. • Intuisjon: ○ Etter første iterasjon er de første to elementene sortert. ○ Etter andre iterasjon er de første tre elementene sortert. ○ Prosessen fortsetter til hele arrayet er sortert. • Egenskaper: ○ Stabil: Elementer med samme nøkkel beholder rekkefølgen. ○ In-place: Krever ingen ekstra datastruktur. ○ Effektiv på små eller nesten sorterte datasett, men dårlig for store, usorterte datasett. • Visualisering: ○ Sortert del → [0 ... i-1] ○ Usortert del → [i ... n-1] ○ Hvert element flyttes fra usortert til sortert del på riktig posisjon.

Heapsort – Idé • Grunnidé: ○ Heapsort kombinerer konseptene heap og sortering. ○ Vi bruker en max-heap, der roten alltid inneholder det største elementet. ○ Ved å bygge en max-heap og deretter "poppe" det største elementet til slutten av arrayet, kan vi sortere hele arrayet i-place. • Trinnvis beskrivelse: 1. Bygg en max-heap av arrayet: § Dette sikrer at hvert foreldre-element er større enn sine barn. § Arrayet kan omformes til heap på O(n) tid ved å "heapify" fra midten og bakover. 2. Sett i = n-1 (siste indeks i arrayet). 3. Flytt roten til slutten av arrayet (A[0] ↔ A[i]) – det største elementet er nå på riktig plass. 4. Reduser heap-størrelsen med 1 (nå er i elementet "utenfor heapen"). 5. Heapify roten for å gjenopprette max-heap-egenskapen. 6. Gjenta trinn 3–5 til i = 0. • Intuisjon: ○ Max-heapen gir alltid tilgang til det største elementet i konstant tid (O(1) for å hente). ○ Flytting av roten til slutten av arrayet og heapify tar O(log n) per element. ○ Totalt: O(n log n) for hele sorteringen. • Egenskaper: ○ In-place: Krever ingen ekstra datastruktur. ○ Ikke stabil: Elementer med samme nøkkel kan endre rekkefølge. ○ Alltid O(n log n), uavhengig av input. ○ God for store datasett når stabilitet ikke er et krav. • Visualisering: ○ Heapbygging: array → max-heap. ○ Sortering: flytt roten til slutten, heapify igjen, fortsett. ○ Etter siste iterasjon er hele arrayet sortert i stigende rekkefølge.

Merge Sort – Idé • Grunnidé: ○ Merge sort er en rekursiv “del og hersk”-algoritme. ○ Vi deler arrayet i to omtrent like store deler, sorterer delene rekursivt, og fletter dem sammen til et sortert array. • Trinnvis beskrivelse: 1. La n være størrelsen på arrayet A. 2. Hvis n ≤ 1, er arrayet allerede sortert – returner A. 3. Beregn midtpunkt: i = ⌊n / 2⌋. 4. Del arrayet i to deler: A[0..i-1] og A[i..n-1]. 5. Kall MergeSort rekursivt på begge delene. 6. Flett de to sorterte delene sammen med Merge-prosedyren. • Intuisjon: ○ Ved å kontinuerlig dele arrayet i mindre biter, blir hvert del-array trivielt å sortere. ○ Fletting av to sorterte arrays kan gjøres lineært i antall elementer.

Merge – sortert fletting • Input: To sorterte arrays A1 og A2. • Output: Et nytt sortert array A med alle elementene. Trinn:

1. Start med pekere i = 0 (A1) og j = 0 (A2).
2. Sammenlign elementene A1[i] og A2[j].
3. Den minste legges inn i A[i+j].
4. Øk tilsvarende peker (i eller j).
5. Fortsett til ett av arraysene er tomt.
6. Kopier eventuelle rester fra det andre arrayet til A. • Kompleksitet: ○ Hver iterasjon øker enten i eller j med 1. ○ Totalt n = |A1| + |A2| iterasjoner → O(n).

Merge Sort – Implementasjon Procedure MergeSort(A) if n ≤ 1 then return A i ← ⌊n/2⌋ A1 ← MergeSort(A[0..i-1]) A2 ← MergeSort(A[i..n-1]) return Merge(A1, A2, A) • Notasjon: ○ A[0..i-1] lager et nytt array med elementene A[0] til A[i-1]. ○ Merge sort kalles rekursivt på hver halvdel, deretter flettes resultatene sammen.

Kjøretidsanalyse • Merge: ○ Itererer gjennom alle elementene i A1 og A2 → O(n). • MergeSort: ○ Halverer arrayet ved hvert rekursivt kall → log(n) nivåer. ○ På hvert nivå gjør vi O(n) arbeid for å flette → O(n log(n)) totalt. • Egenskaper: ○ Stabil: Bevarer rekkefølgen til elementer med samme nøkkel. ○ Ikke in-place i denne implementasjonen, fordi vi lager nye arrays for delene. ○ Alltid O(n log n) i beste, gjennomsnittlige og verste tilfelle.

Quicksort – Idé • Grunnidé: Quicksort er en effektiv rekursiv "del og hersk"-algoritme for sortering. ○ Vi velger et pivot-element i arrayet. ○ Alle elementer mindre enn pivot flyttes til venstre. ○ Alle elementer større enn pivot flyttes til høyre. ○ Vi sorterer deretter venstre og høyre del rekursivt. • Trinnvis beskrivelse: 1. Velg pivot-element A[i] der $0 \leq i < n$. 2. Start to pekere: § left fra starten av arrayet mot høyre, søker elementer større enn pivot. § right fra slutten av arrayet mot venstre, søker elementer mindre enn pivot. 3. Bytt plass på elementene som left og right peker på dersom de er på feil side. 4. Fortsett søket etter nye elementer som må byttes. 5. Avslutt når left og right krysses – pivot er nå på sin endelige posisjon. 6. Gjenta prosessen rekursivt på array-delen til venstre og til høyre for pivot. • Intuisjon: ○ Etter hver partisjonering er pivot på riktig plass. ○ Problemstørrelsen halveres rekursivt, som gir potensielt log(n) nivåer av rekursjon.

Quicksort – Valg av pivot • Hvorfor pivot er viktig: ○ Effektiviteten til quicksort avhenger sterkt av valg av pivot. ○ Ideelt pivot-element: medianen i arrayet → balanserte del-arrayer. ○ Problem: Å finne median krever sortering → gir ikke gevinst. • Vanlige pivot-strategier: 1. Tilfeldig valg: Enkelt og gir gjennomsnittlig gode resultater. 2. Median av tre: Median av A[0], $A[\lfloor n/2 \rfloor]$ og A[n-1]. § Reduserer sjansen for dårlig ytelse på nesten sorterte arrayer. 3. Første eller siste element: § Kan gi verstefall $O(n^2)$ dersom arrayet allerede er sortert eller nesten sortert. • Notasjon i algoritmer: ○ Vi antar en funksjon ChoosePivot(A) som velger pivot basert på en rimelig strategi.

Kjøretid og egenskaper • Gjennomsnittlig tilfelle: ○ Hvert element besøkes omtrent én gang per nivå i rekursjon → O(n log n). • Verstefall: ○ Skje når pivot alltid velges dårlig (f.eks. minste eller største element) → $O(n^2)$. • Stabilitet: ○ Quicksort er ikke stabil uten ekstra arbeid. • In-place: ○ Standard quicksort kan implementeres in-place uten ekstra datastruktur.

Quicksort – Partition • Formål: Partition-funksjonen sørger for at alle elementer mindre enn pivot plasseres til venstre, og alle elementer større enn pivot plasseres til høyre. Pivot ender på sin endelige posisjon. • Trinnvis forklaring av algoritmen: 1. Velg et pivot-element p ved hjelp av ChoosePivot(A, low, high). 2. Bytt pivot til enden av arrayet (A[high]) for enklere håndtering. 3. Initialiser pekere: § left = low § right = high - 1 4. Løkke: Så lenge left ≤ right: § Flytt left mot høyre så lenge A[left] ≤ pivot. § Flytt right mot venstre så lenge A[right] ≥ pivot. § Hvis left < right, bytt A[left] og A[right] for å rette opp feilplasserte elementer. 5. Etter løkken bytter vi pivot (A[high]) med A[left]. 6. Returner left som pivotens endelige posisjon. • Intuisjon: ○ Pivot er nå på riktig plass. ○ Alle til venstre for pivot er ≤ pivot, alle til høyre er ≥ pivot. ○ Resten sorteres rekursivt.

Quicksort – Implementasjon • Hovedalgoritme: 1. Sjekk basis-tilfelle: low ≥ high, da er arrayet tomt eller kun ett element → allerede sortert. 2. Kall Partition(A, low, high) for å plassere pivot riktig og få dens indeks p. 3. Rekursivt kall: § Sorter venstre del: Quicksort(A, low, p - 1) § Sorter høyre del: Quicksort(A, p + 1, high) 4. Returner arrayet. • Sortering av hele arrayet: ○ Kall: Quicksort(A, 0, n - 1)

Quicksort – Kjøretidsanalyse • Partition: ○ Går gjennom alle elementer i den gjeldende delen av arrayet → O(high – low) lineær tid. • Rekursjon: ○ Den gjenværende array-delen blir mindre etter hver partisjonering. ○ Beste tilfelle: Pivot alltid "midten" → halvering av array → O(n log n) totalt. ○ Verste tilfelle: Pivot alltid ytterst (f.eks. første element i sortert array) → array krymper med 1 hver gang → $O(n^2)$

totalt. • Gjennomsnittlig tilfelle: ○ Tilfeldig pivot eller median-av-tre → gjennomsnittlig O(n log n). • Oppsummering: ○ In-place: ja, krever ingen ekstra array. ○ Stabilitet: nei, bytter elementer rundt pivot. ○ Praktisk ytelse: svært effektivt for de fleste datasett.

Bucket Sort – Introduksjon • Hovedidé: ○ Alle sorteringsalgoritmer vi har sett hittil (Bubble, Selection, Insertion, Heapsort, Merge, Quicksort) er basert på sammenligning. ○ Slike algoritmer kan ikke gjøre bedre enn O(n log n) i gjennomsnitt. ○ Hvis vi vet mer om verdiene, kan vi gjøre bedre. • Hvordan bucket sort fungerer: 1. Lag N bøtter, hvor hver bøtte representerer en kategori eller nøkkelområde. 2. Hvert element har en nøkkel (kategori) som bestemmer hvilken bøtte det havner i. 3. Plasser hvert element i riktig bøtte basert på nøkkelen. 4. Løp gjennom bøttene i rekkefølge og kopier elementene tilbake til arrayet. 5. Noen ganger sorteres elementene i hver bøtte internt (f.eks. med insertion sort), men dette er ikke alltid nødvendig. • Intuisjon: ○ Tenk på nøklene som "adresser" til bøtter. ○ Alle elementer havner i sin riktige kategori uten å sammenligne alle med alle.

Bucket Sort – Eksempel: Kortstokk • Orden på kort: ○ Verdier: A < 2 < 3 < … < 10 < J < Q < K ○ Sorter: ♣ < ♢ < ♠ < ♡ • Mål: ○ Sorter en hånd med kort slik at: § Alle kort med samme sort kommer sammen. § Hver sort er innbyrdes sortert etter verdi. • Metode med bucket sort: 1. Først sorterer vi på verdi (A–K). 2. Deretter sorterer vi på sort (♣, ♢, ♠, ♡). • Merk: Dette illustrerer stabilitet, fordi rekkefølgen innenfor samme nøkkel beholdes etter første sortering.

Bucket Sort – Implementasjon ALGORITHM: BUCKET SORT Input: Array A med n elementer Output: Sortert array A 1 Procedure BucketSort(A) 2 La B være et array med N tomme lister 3 for i ← 0 to n–1 do 4 La k være nøkkelen assosiert med A[i] 5 Legg til A[i] på slutten av listen B[k] 6 j ← 0 7 for k ← 0 to N–1 do 8 for hver x i listen B[k] do 9 A[j] ← x 10 j ← j + 1 11 return A • Forklaring: ○ B er en liste av bøtter. ○ Hver element A[i] havner i bøtte B[k] basert på nøkkel k. ○ Til slutt samles alle bøttene tilbake i A i riktig rekkefølge.

Bucket Sort – Kjøretidsanalyse • Trinn 1: Plasser alle elementer i bøttene → O(n). • Trinn 2: Gå gjennom alle bøttene og kopier elementer tilbake → O(N) + O(n). • Trinn 3: Hvis N er konstant (f.eks. 52 kort), forenkles dette til O(n). • Oppsummering: ○ Bucket sort er lineær i beste tilfelle hvis antall bøtter N er liten eller konstant. ○ Krever at nøklene er begrenset til et kjent intervall. ○ Stabil sorteringsalgoritme dersom vi bruker stabil sortering innen bøttene.

Radix Sort – Introduksjon • Hovedidé: ○ Radix sort er nært beslektet med bucket sort, og kan betraktes som suksessiv anvendelse av bucket sort på forskjellige deler av nøkkelen. ○ Eksempelet med kortstokk (sortering først på verdi, deretter på sort) er faktisk en type radix sort. ○ Radix sort brukes for data som kan ordnes leksikografisk, dvs. der elementene kan sammenlignes siffer for siffer eller bokstav for bokstav. • Intuisjon: ○ Vi "sorterer etter plassverdi" fra minst signifikant til mest signifikant, eller omvendt, og oppnår en fullstendig sortering.

Radix Sort – Leksikografiske ordninger • Definisjon: ○ En leksikografisk ordning er en generalisering av alfabetisk rekkefølge. ○ Ord eller tall ordnes slik at første symbol prioriteres over andre, som prioriteres over tredje, osv. • Formell definisjon: ○ Gitt to tupler (a1,a2,…,ad) og (b1,b2,…,bd): § (a1,…,ad)<(b1,…,bd) hvis: ☐ a1<b1 eller ☐ a1=b1 og (a2,…,ad)<(b2,…,bd) • Eksempel med kort: ○ 7♣ < J♣ fordi: § Sorten ♣ = ♣ § Men 7 < J • Eksempel med tall: ○ Sortering skjer siffer for siffer, fra minst til mest signifikant (eller omvendt avhengig av implementasjon).

Radix Sort – Eksempel med heltall • Gitt tall: 1814, 232, 2888, 31, 1455, 2242, 4345, 1470, 515, 3632 • Fremgangsmåte: 1. Pad tallene med nuller for like mange siffer: 1814, 0232, 2888, 0031, 1455, 2242,

4345, 1470, 0515, 3632 2. Sorter minste siffer først (bucket sort). 3. Deretter neste siffer, og så videre mot mest signifikante siffer. • Etter alle siffer er sortert: 0031, 0232, 0515, 1455, 1470, 1814, 2242, 2888, 3632, 4345 • Merk: ○ Hver passering med bucket sort er stabil, noe som gjør at rekkefølgen for tidligere siffer beholdes.

Radix Sort – Implementasjon for positive heltall ALGORITHM: RADIX SORT FOR POSITIVE HELTALL Input: Array A med n positive heltall Output: Sortert array A 1 Procedure RadixSort(A) 2 d ← antall siffer i det største tallet 3 for i ← d–1 down to 0 do 4 A ← BucketSort(A) etter det i-te sifferet 5 return A • Forklaring: ○ Vi starter med minst signifikante siffer (LSB) eller mest signifikante (MSB) avhengig av variant. ○ Bucket sort brukes på hvert siffer. ○ Etter alle passeringer blir arrayet sortert fullstendig.

Radix Sort – Kjøretidsanalyse • Bucket sort: O(N + n), der N = antall mulige nøkkelverdier (f.eks. 10 for sifre). • Radix sort: Gjør d antall bucket sort-passeringer (d = antall siffer). • Totalt blir det: O(d _ (N + n)) • Eksempel med heltall: ○ N = 10 (titallsystemet), d ≤ 9 for tall < 10^9 ○ Kjøretid: O(d _ n) ≈ O(n) for faste størrelser av d og N • Oppsummering: ○ Radix sort kan være lineær tid for store datasett med begrensede nøkkelverdier. ○ Krever stabil bucket sort. ○ Effektiv for heltall, strenger, eller andre leksikografiske data.