Programmazione Avanzata e parallela

Lezione 9

Luca Manzoni

Rivisitare la ricerca binaria

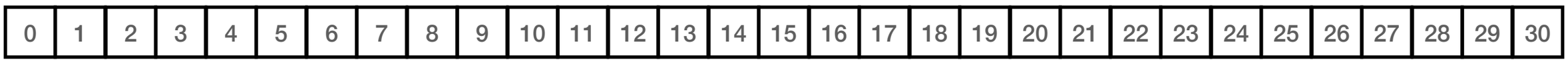
Ricerca binaria

Disposizione in memoria

- Rivisitiamo la ricerca binaria vedendo l'effetto sulla cache
- Caso interessante perché ha due pattern differenti:
  - Indirizzi di memoria con buona località spaziale
  - Indirizzi di memoria con buona località temporale
- Possiamo trovare una disposizione in memoria più efficiente?

Ricerca binaria

Pattern di accesso



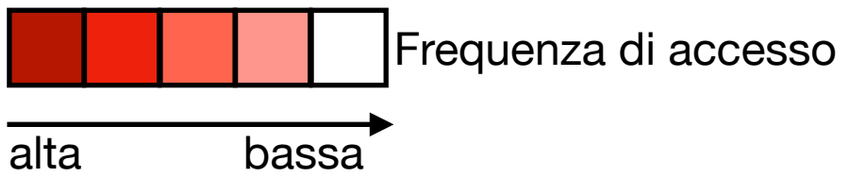
- Consideriamo due fattori che influenzano la presenza di valori nella cache quando effettuiamo una ricerca binaria su un array

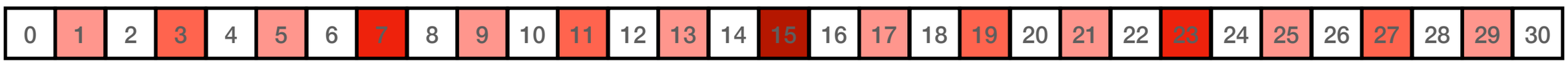
- **Località spaziale**: quando accediamo a valori vicini in memoria?

- **Località temporale**: ci sono valori a cui accediamo spesso?

Ricerca binaria

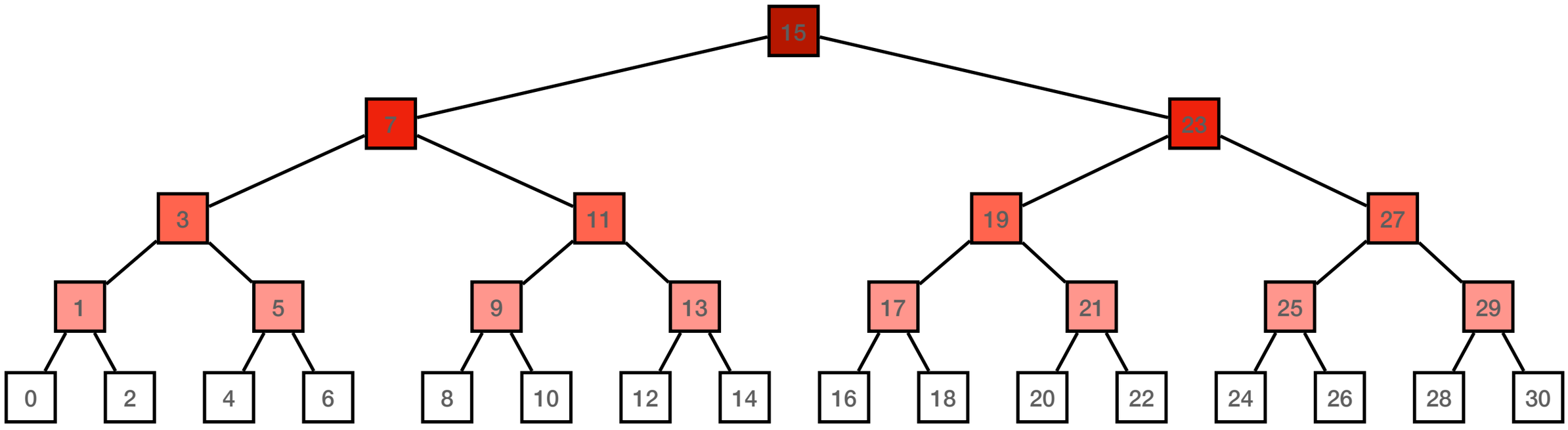
Pattern di accesso: località temporale





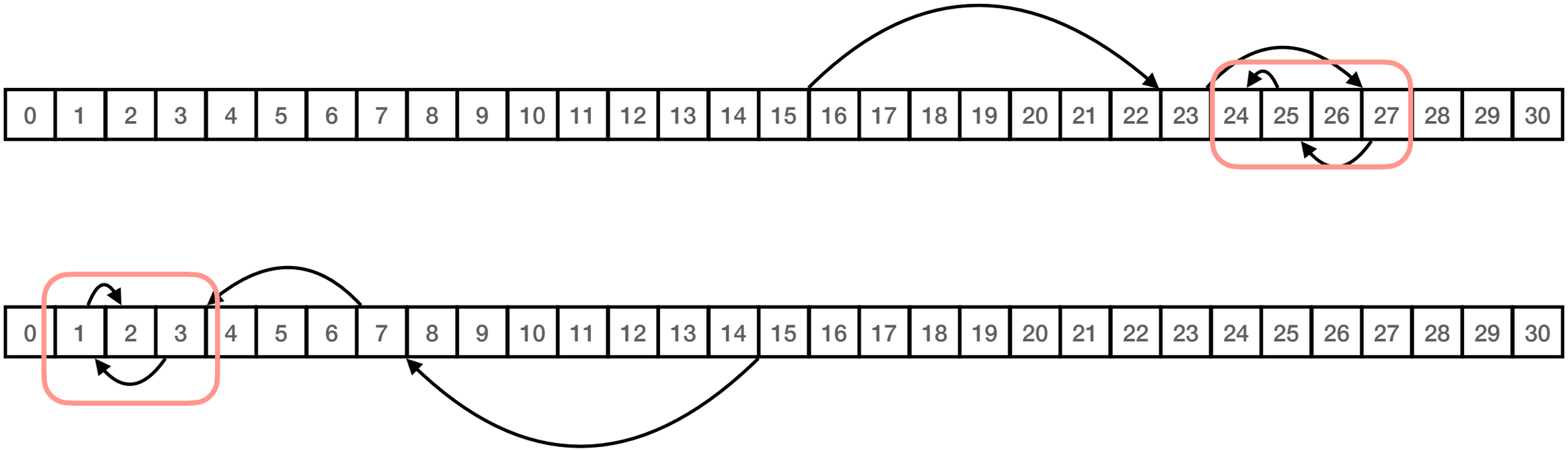
I valori “al centro” sono visti più di frequente perché “ci passiamo” per ogni ricerca

Questo è più evidente se rappresentiamo la ricerca come un albero:



Ricerca binaria

Pattern di accesso: località spaziale



- Notiamo che verso la fine della ricerca gli intervalli da cercare sono piccoli
- Questo garantisce una buona località spaziale, dato che sanno valori caricati nella stessa cache line
- All'inizio della ricerca la località spaziale è invece molto bassa: "saltiamo" a valori molto distanti in memoria che non saranno già caricati in cache

Ricerca binaria

Disposizione in memoria

- Possiamo trovare una disposizione in memoria dell'array che sia più favorevole alla cache?

- L'array ordinato possiamo vederlo come un albero binario di n nodi in cui:

  - La radice è in posizione $n/2$

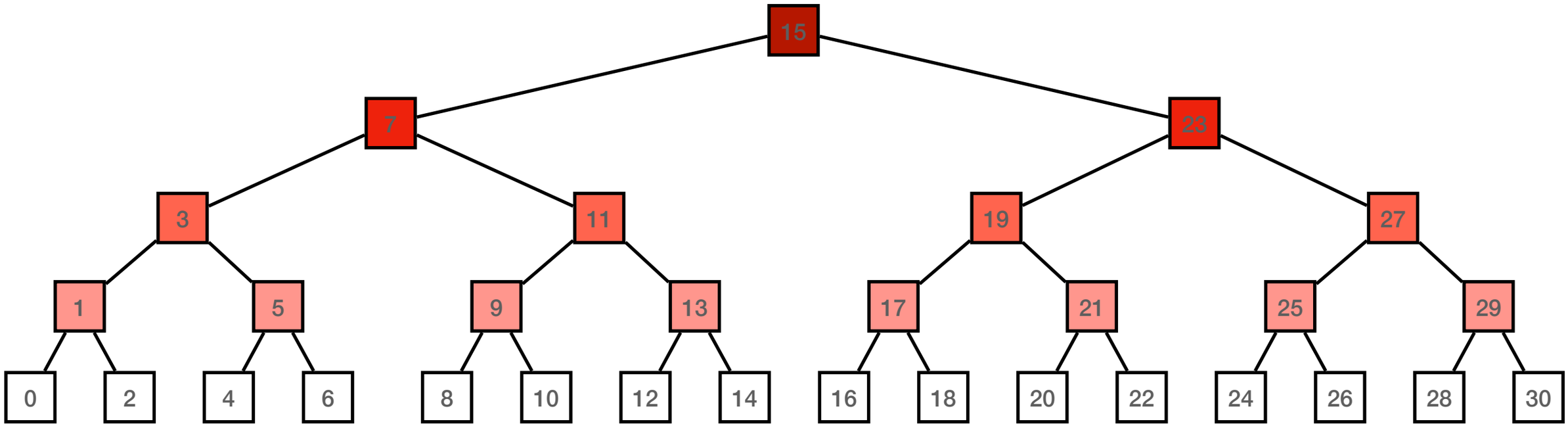
  - I figli della radice sono in posizione $n/4$ e $3n/4$

  - etc.

- Esiste un altro modo di codificare lo stesso albero?

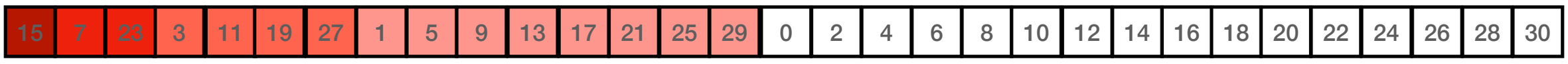
Ricerca binaria

Utilizzo di un albero di ricerca senza puntatori



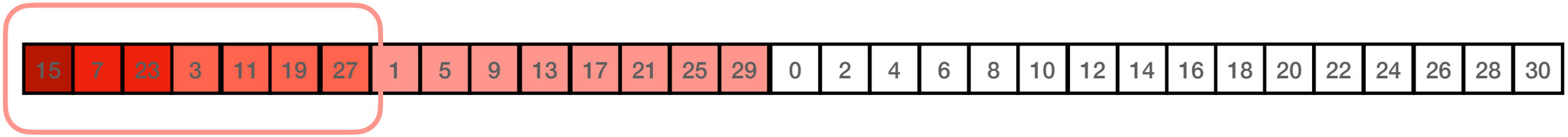
Possiamo rappresentare un albero di ricerca in un array in cui

- La radice è in posizione 1 (indicizzazione a partire da 1)
- I figli del nodo in posizione i sono agli indici $2i$ e $2i + 1$



Ricerca binaria

Utilizzo di un albero di ricerca senza puntatori



Buona località temporale e spaziale:
la parte dell'albero vicina alla radice viene subito messa tutta in cache

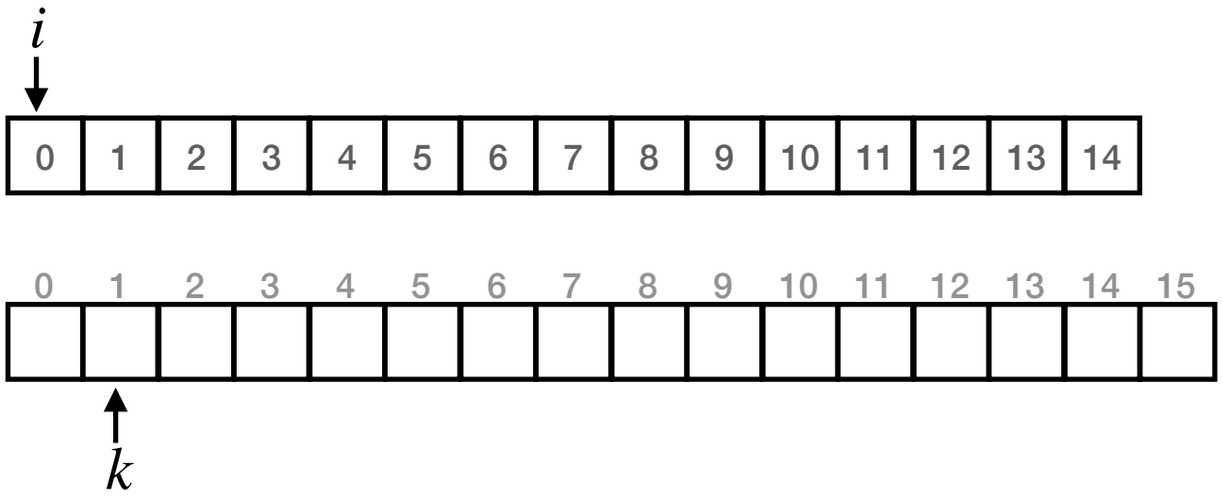
Ricerca binaria

Costruire da un array ordinato

- Array iniziale a di n elementi già ordinato
- Per semplicità il nostro array target t avrà $n + 1$ elementi, con quello di indice 0 non usato
- Dato un indice i e una posizione k nell'array target
  - Scrivi ricorsivamente i valori (a partire da i) a partire dalla posizione $2k$ a sinistra di quello di mezzo aggiornando i
  - Scrivi il valore di mezzo (dato dal nuovo valore di i)
  - Scrivi ricorsivamente i valori rimanenti a partire dalla posizione $2k + 1$

Ricerca binaria

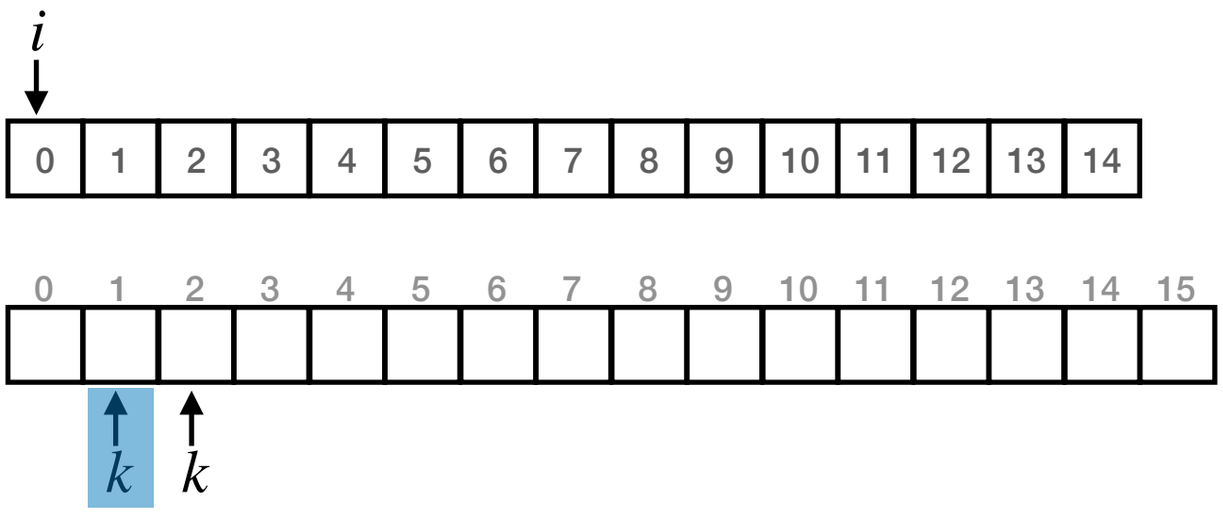
Costruire da un array ordinato




```
int reorder_array(int * a, int * t, int len, int k, int i) {
  if (k <= len) {
    i = reorder_array(a, t, len, 2 * k, i);
    t[k] = a[i++];
    i = reorder_array(a, t, len, 2 * k + 1, i);
  }
  return i;
}
```

Ricerca binaria

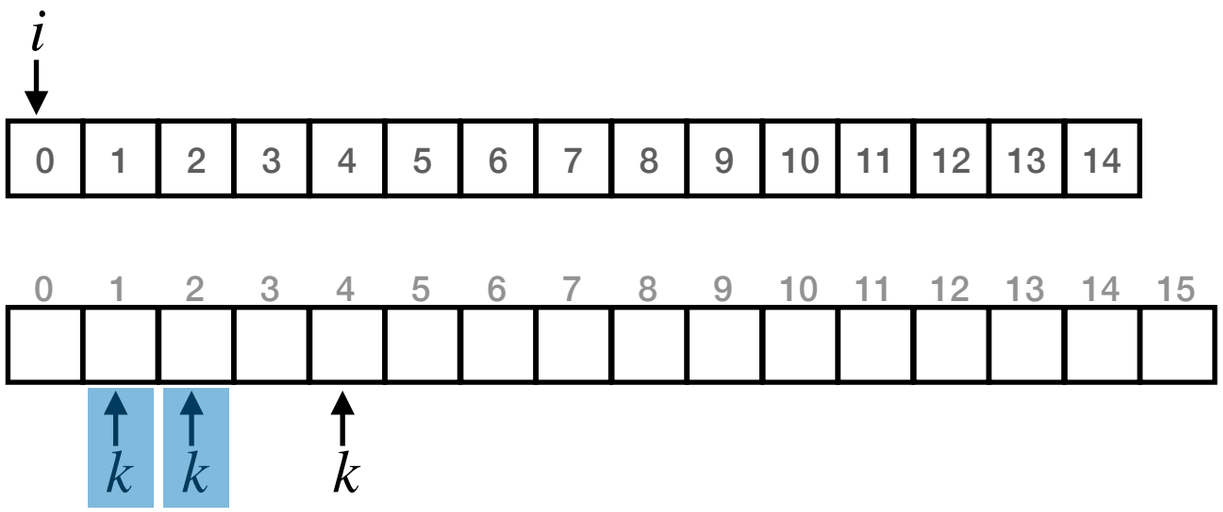
Costruire da un array ordinato




```c
int reorder_array(int * a, int * t, int len, int k, int i) {
  if (k <= len) {
    i = reorder_array(a, t, len, 2 * k, i);
    t[k] = a[i++];
    i = reorder_array(a, t, len, 2 * k + 1, i);
  }
  return i;
}
```

Ricerca binaria

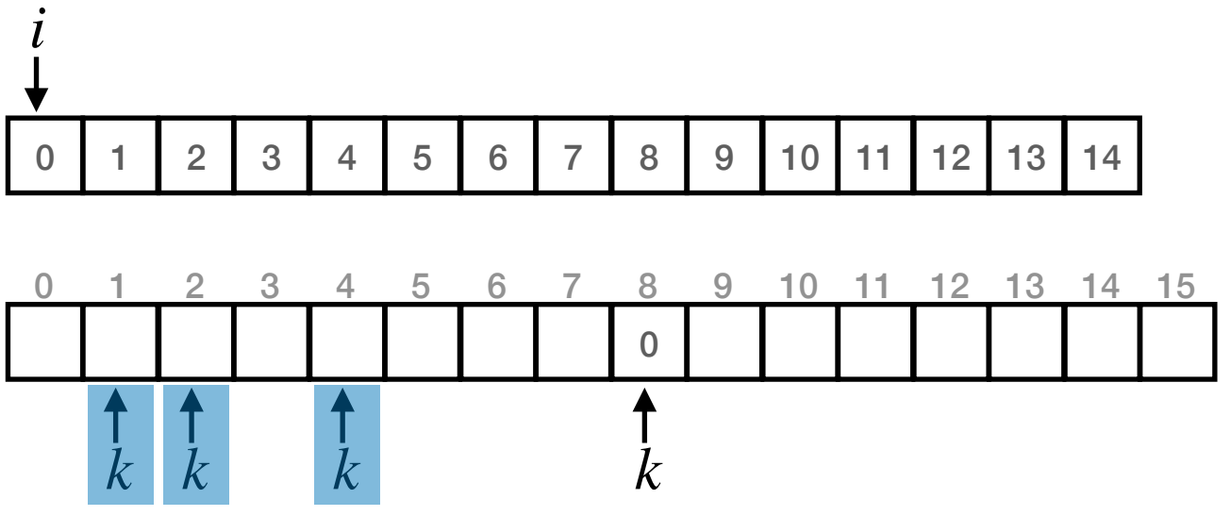
Costruire da un array ordinato




```
int reorder_array(int * a, int * t, int len, int k, int i) {
  if (k <= len) {
    i = reorder_array(a, t, len, 2 * k, i);
    t[k] = a[i++];
    i = reorder_array(a, t, len, 2 * k + 1, i);
  }
  return i;
}
```

Ricerca binaria

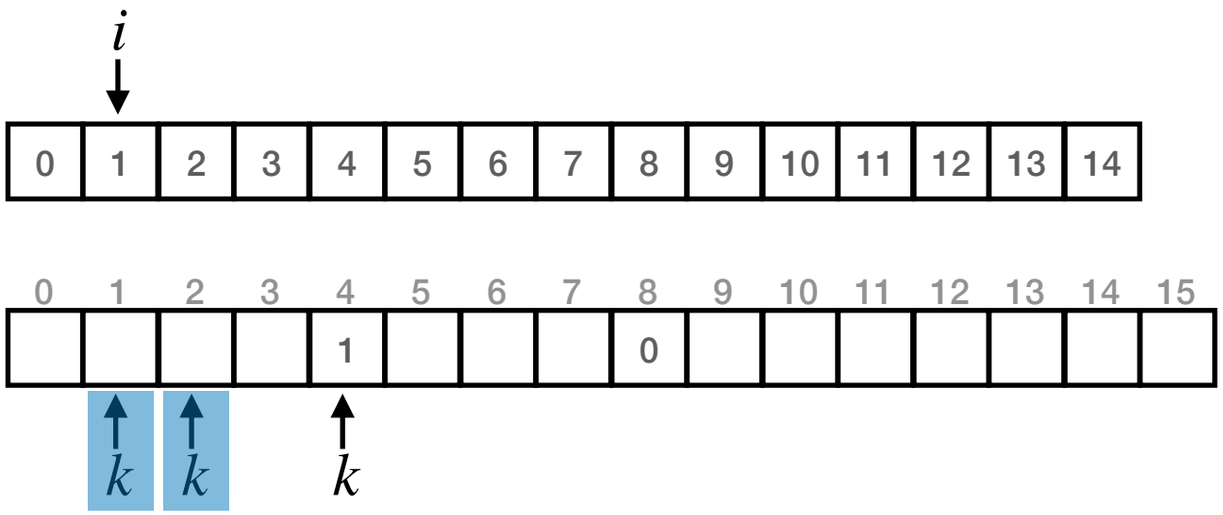
Costruire da un array ordinato




```
int reorder_array(int * a, int * t, int len, int k, int i) {
  if (k <= len) {
    i = reorder_array(a, t, len, 2 * k, i);
    t[k] = a[i++];
    i = reorder_array(a, t, len, 2 * k + 1, i);
  }
  return i;
}
```

Ricerca binaria

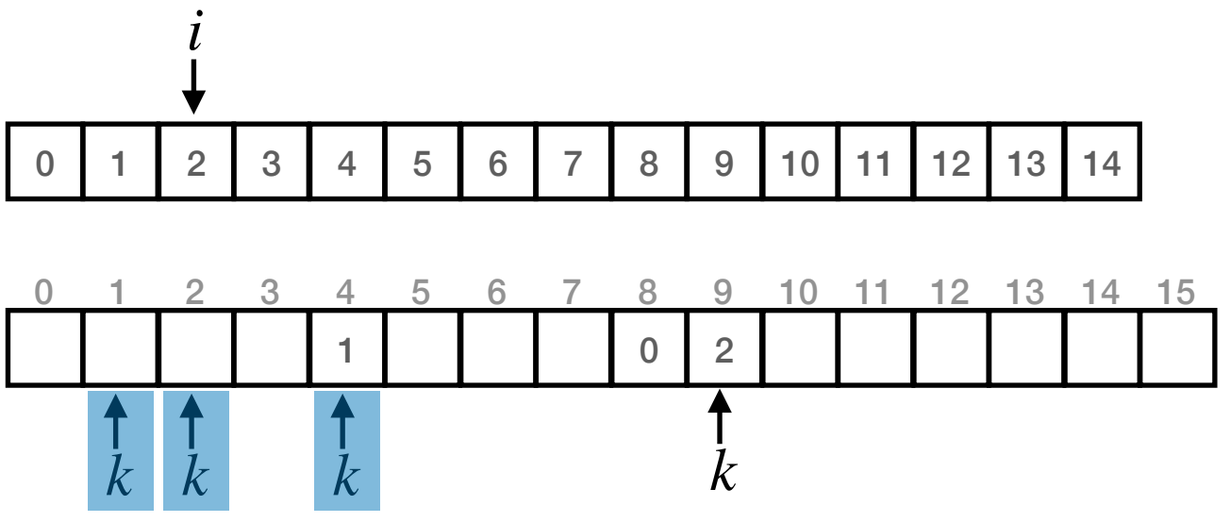
Costruire da un array ordinato




```
int reorder_array(int * a, int * t, int len, int k, int i) {
  if (k <= len) {
    i = reorder_array(a, t, len, 2 * k, i);
    t[k] = a[i++];
    i = reorder_array(a, t, len, 2 * k + 1, i);
  }
  return i;
}
```

Ricerca binaria

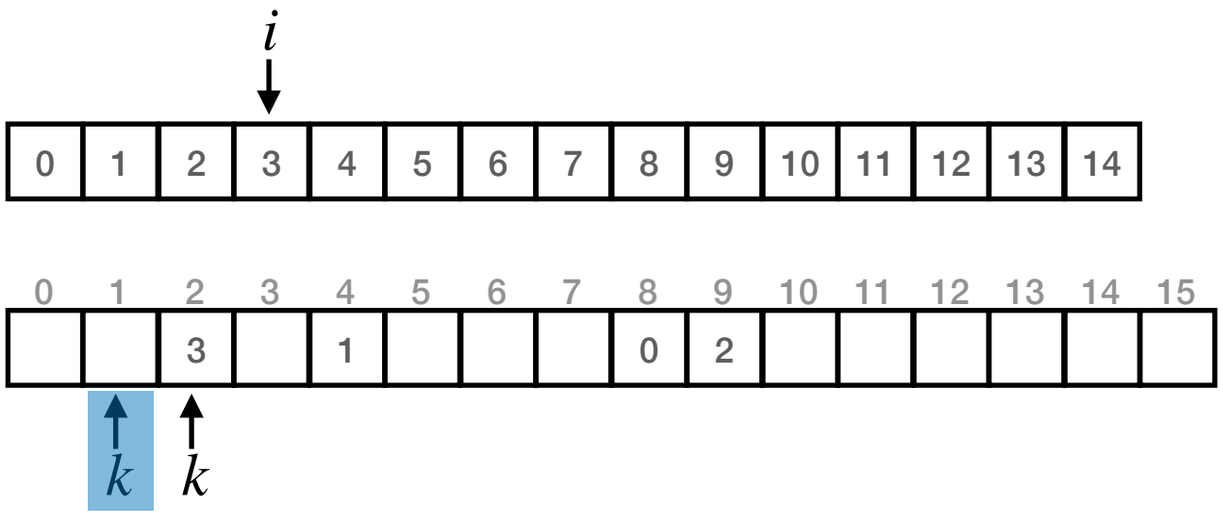
Costruire da un array ordinato




```
int reorder_array(int * a, int * t, int len, int k, int i) {
  if (k <= len) {
    i = reorder_array(a, t, len, 2 * k, i);
    t[k] = a[i++];
    i = reorder_array(a, t, len, 2 * k + 1, i);
  }
  return i;
}
```

Ricerca binaria

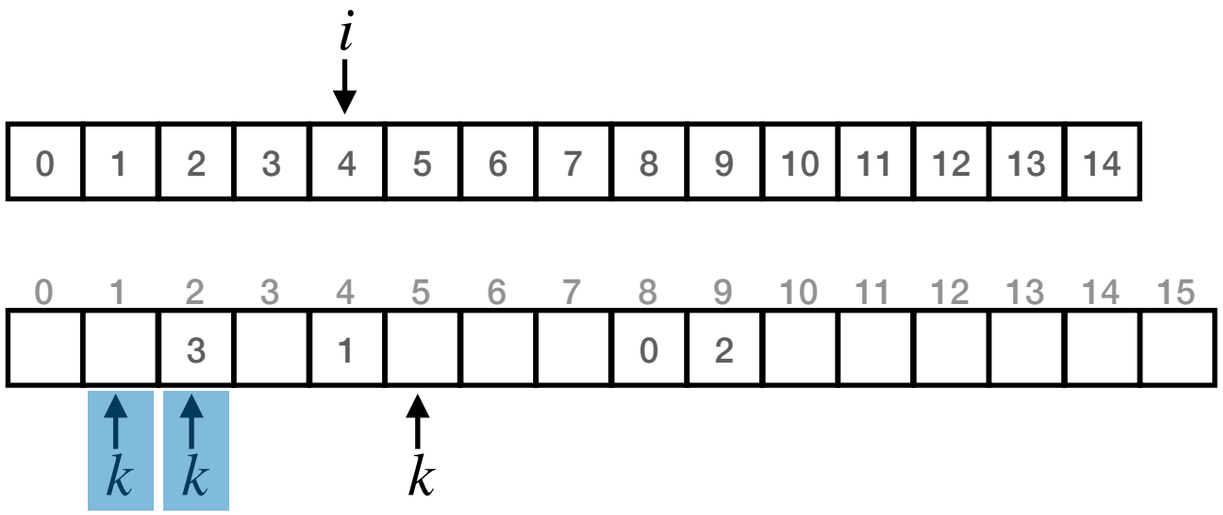
Costruire da un array ordinato




```
int reorder_array(int * a, int * t, int len, int k, int i) {
  if (k <= len) {
    i = reorder_array(a, t, len, 2 * k, i);
    t[k] = a[i++];
    i = reorder_array(a, t, len, 2 * k + 1, i);
  }
  return i;
}
```

Ricerca binaria

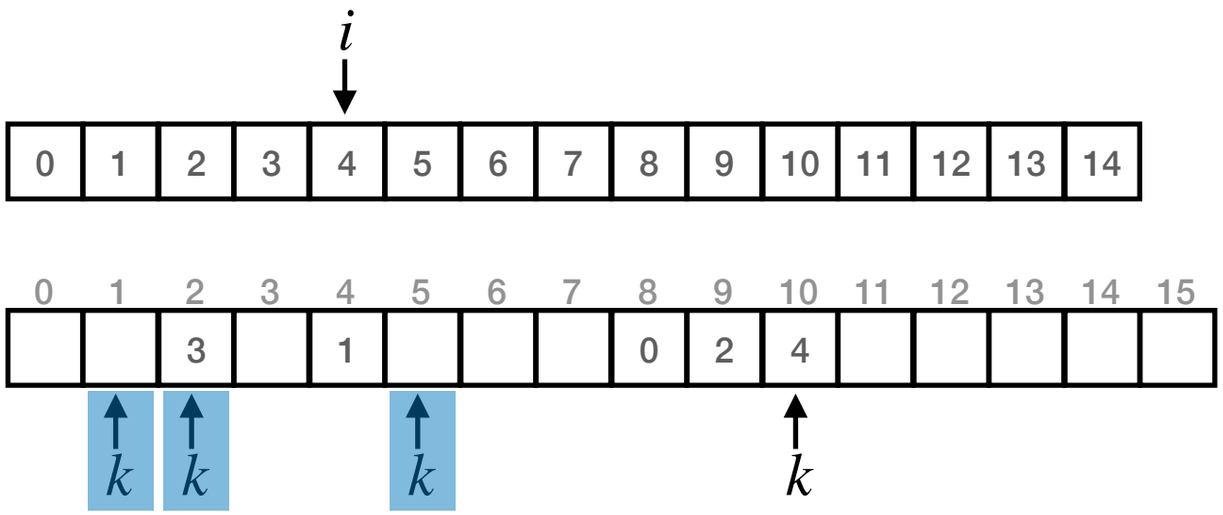
Costruire da un array ordinato




```
int reorder_array(int * a, int * t, int len, int k, int i) {
  if (k <= len) {
    i = reorder_array(a, t, len, 2 * k, i);
    t[k] = a[i++];
    i = reorder_array(a, t, len, 2 * k + 1, i);
  }
  return i;
}
```

Ricerca binaria

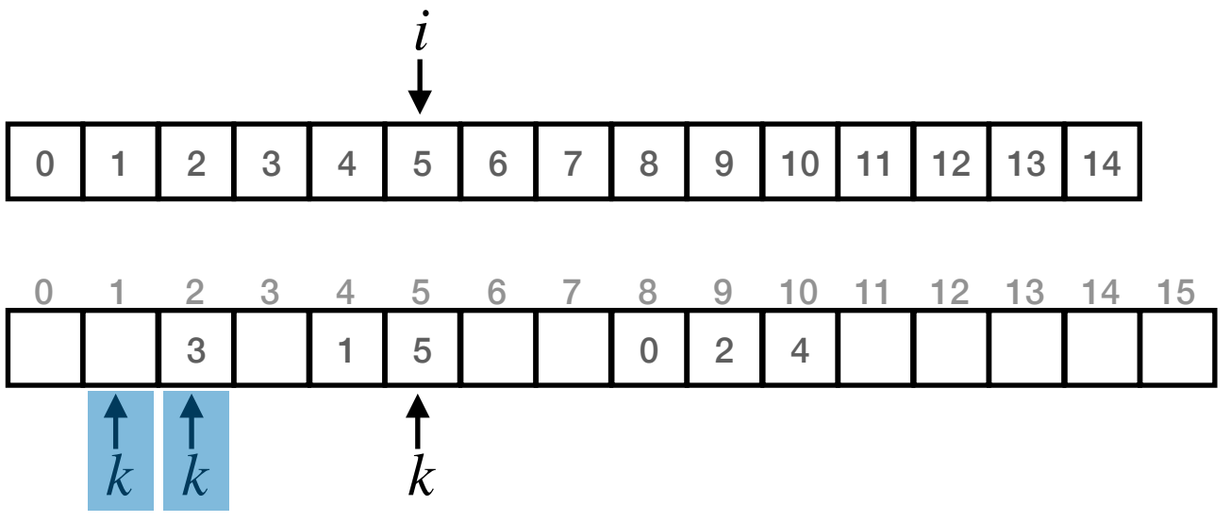
Costruire da un array ordinato




```
int reorder_array(int * a, int * t, int len, int k, int i) {
  if (k <= len) {
    i = reorder_array(a, t, len, 2 * k, i);
    t[k] = a[i++];
    i = reorder_array(a, t, len, 2 * k + 1, i);
  }
  return i;
}
```

Ricerca binaria

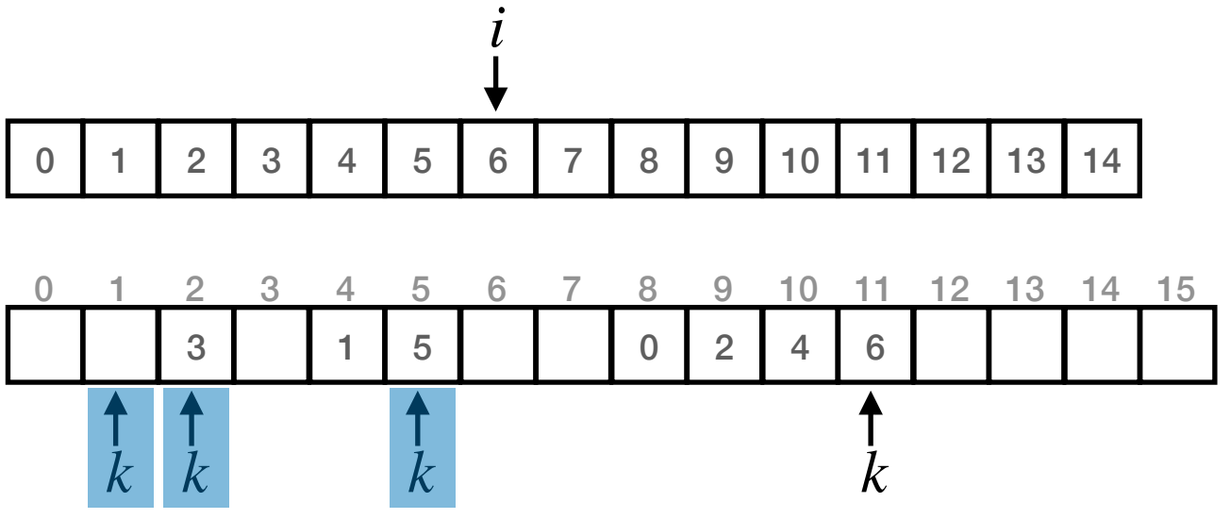
Costruire da un array ordinato




```
int reorder_array(int * a, int * t, int len, int k, int i) {
  if (k <= len) {
    i = reorder_array(a, t, len, 2 * k, i);
    t[k] = a[i++];
    i = reorder_array(a, t, len, 2 * k + 1, i);
  }
  return i;
}
```

Ricerca binaria

Costruire da un array ordinato




```
int reorder_array(int * a, int * t, int len, int k, int i) {
  if (k <= len) {
    i = reorder_array(a, t, len, 2 * k, i);
    t[k] = a[i++];
    i = reorder_array(a, t, len, 2 * k + 1, i);
  }
  return i;
}
```

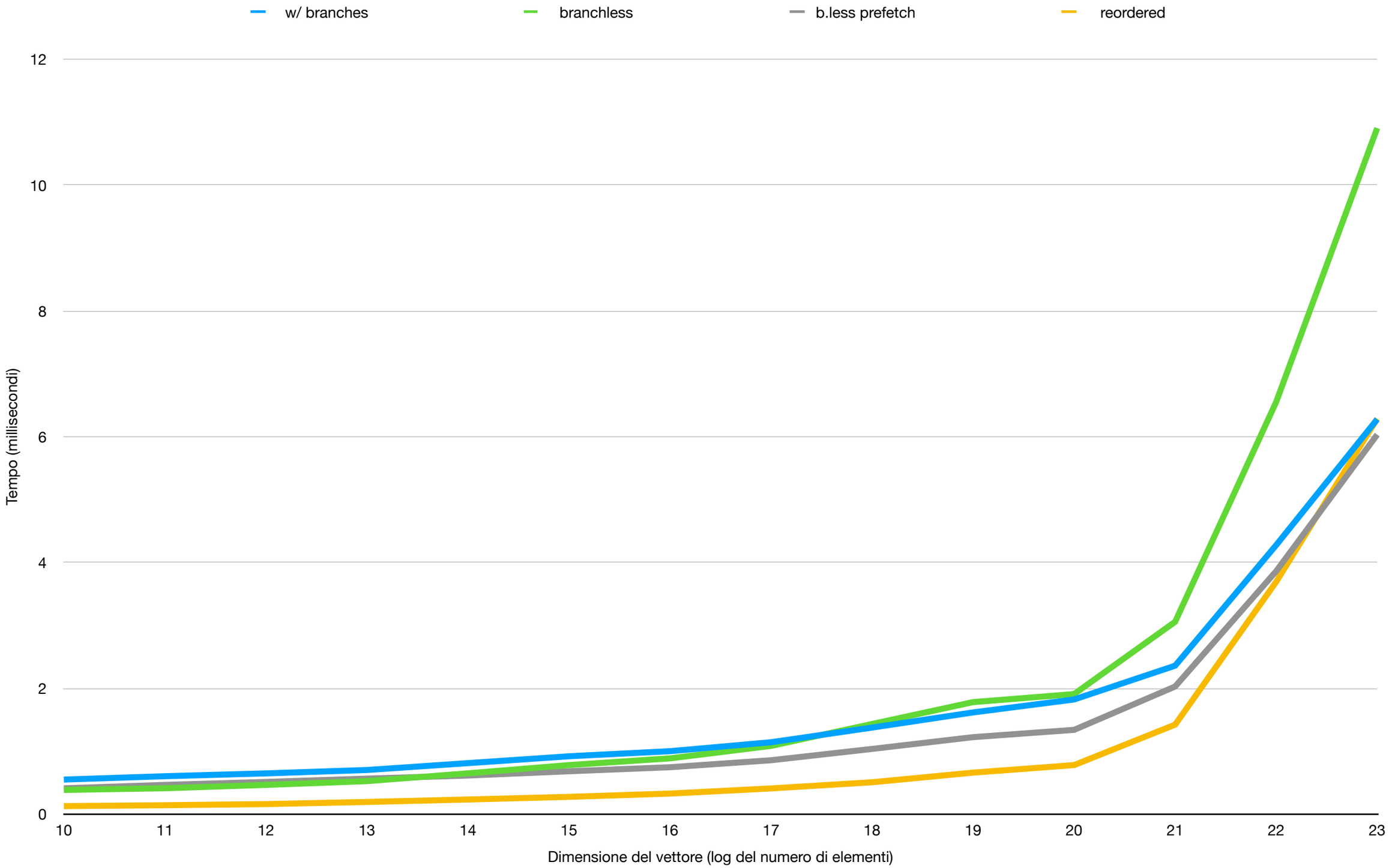
Ricerca binaria

Fare la ricerca

- La ricerca può avvenire come una vista in un albero binario
- A partire dalla posizione $k = 1$
  - Se la chiave è minore allora andiamo nell'albero di sinistra, impostando l'indice a $k \leftarrow 2k$
  - Altrimenti andiamo nel sottoalbero di destra, impostando l'indice a $k \leftarrow 2k + 1$
- Terminiamo quando l'indice supera la fine dell'array

Ricerca “riordinata”

Primi risultati



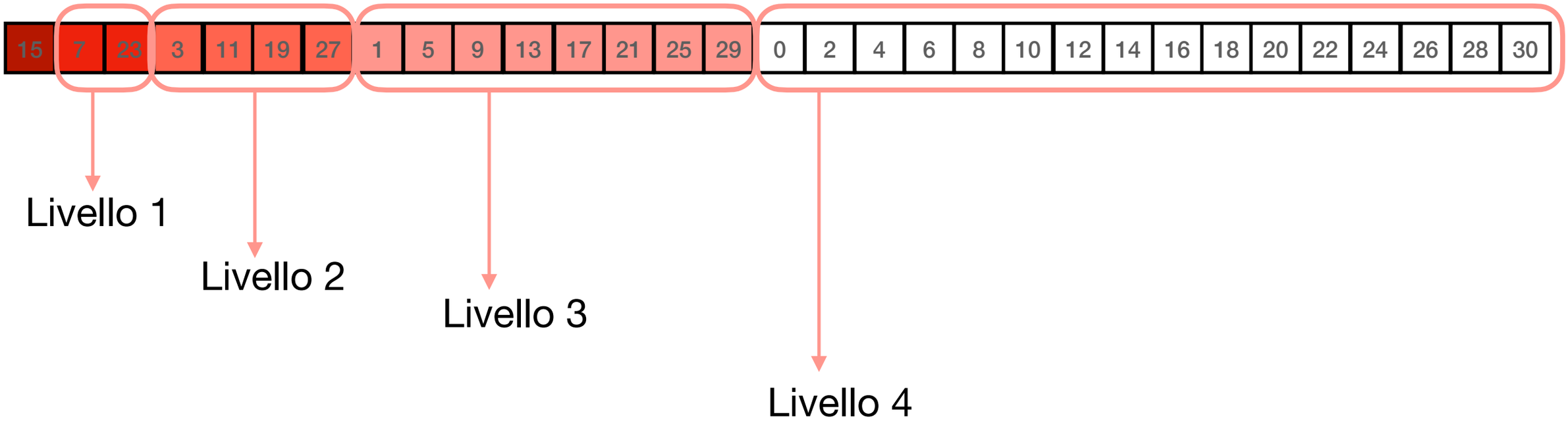
Ricerca binaria

Primi risultati

- La situazione è buona per valori piccoli
- Peggiora per valori grandi
- Rispetto alla ricerca binaria normale abbiamo che la parte finale della ricerca non ha una buona località spaziale
- Per un indice k i figli saranno in posizione $2k$ e $2k + 1$…
- …che per k elevato significa che non sono nella stessa cache line
- Possiamo migliorare la situazione?

Ricerca binaria

Utilizzo di un albero di ricerca senza puntatori



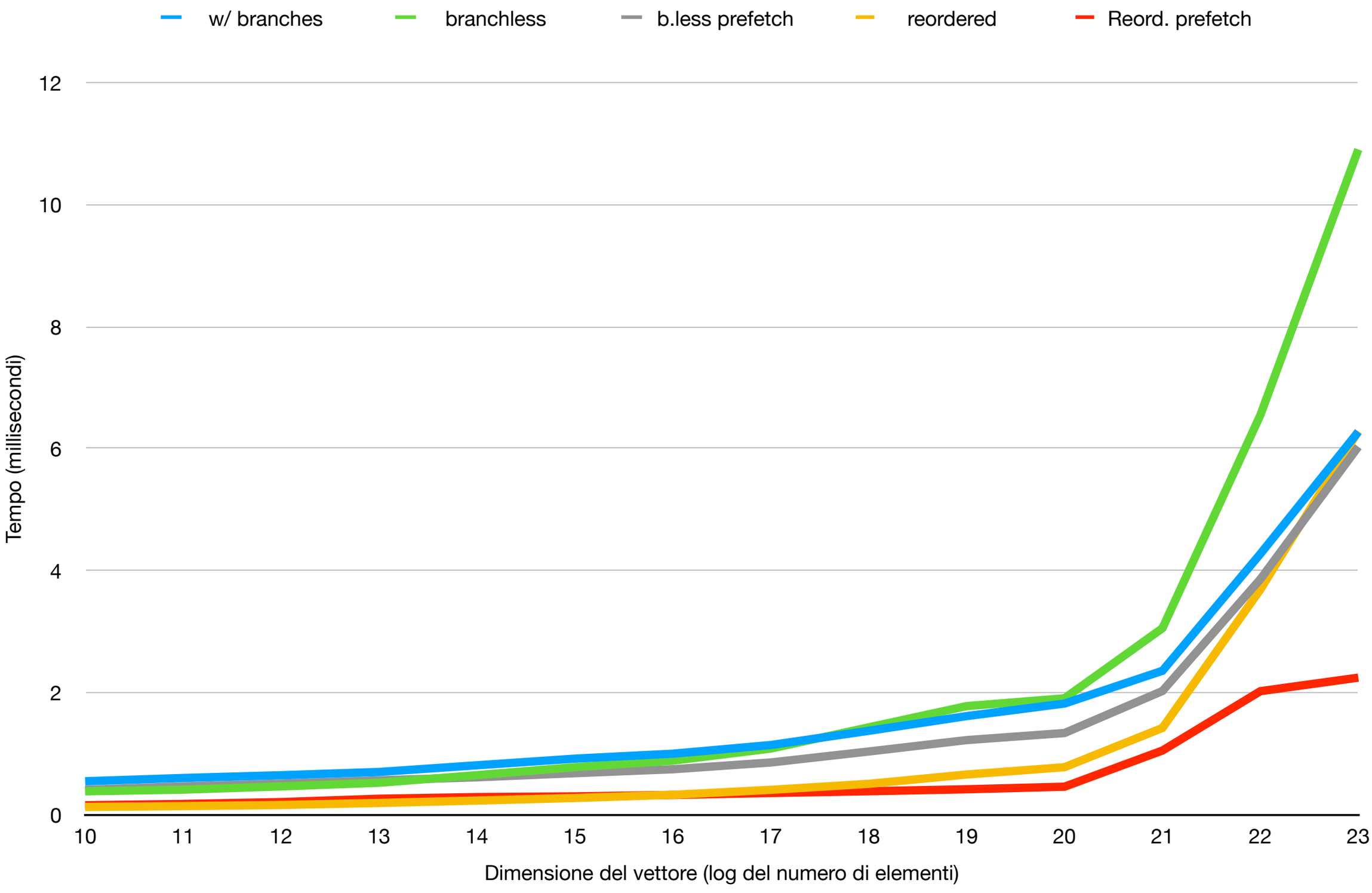
Accediamo sicuramente ad almeno un elemento per livello

Se siamo al livello i il livello $i + 4$ *limitato alla parte che possiamo raggiungere* ha $2^4 = 16$ elementi consecutivi in memoria

Possiamo chiedere di pre-caricare in quel blocco (sapendo che tanto ci accederemo) tramite una operazione di prefetch

Ricerca “riordinata”

Risultati con prefetch



Ricerca binaria

Cosa abbiamo visto?

- Possiamo velocizzare la ricerca utilizzando codice privo di branch difficili da predire
- È importante tenere in considerazione le interazioni con la memoria quando le dimensioni superano quelle della cache
- La località di memoria è importante per le prestazioni
- La capacità di sapere in anticipo a quali indirizzi accederemo permette di pre-caricare il loro contenuto in cache limitando la latenza